

Big Data Processing with Spark

Spark大数据处理

原理、算法与实例

刘　军　林文辉　方　澄◎编著

清华大学出版社

Spark 大数据处理：原理、算法与实例

刘军　林文辉　方澄　编著

清华大学出版社
北京

内 容 简 介

本书以时下最为流行的 Hadoop 所存在的缺陷为出发点，深入浅出地介绍了下一代大数据处理核心技术 Spark 的优势和必要性，并以最简洁的指引步骤展示了如何在 10 分钟内建立一个 Spark 大数据处理环境。在此基础上，以图文并茂和丰富的示例代码讲解的形式系统性地揭示了 Spark 的运行原理、算子使用、算法设计和优化手段，为读者提供了一本快速由浅入深掌握 Spark 基础能力和高级技巧的参考书籍。

本书共 6 章，涉及的主题主要包括大数据处理技术从 Hadoop 发展到 Spark 的必然性、快速体验 Spark 的指引、Spark 架构和原理、RDD 算子使用方法和示例、Spark 算法设计实例、Spark 程序优化方法。

本书适合需要使用 Spark 进行大数据处理的程序员、架构师和产品经理作为技术参考和培训资料，亦可作为高校研究生和本科生教材。

图书在版编目(CIP)数据

Spark 大数据处理：原理、算法与实例/刘军，林文辉，方澄编著. --北京：清华大学出版社，2016
ISBN 978-7-302-44995-9

Ⅰ. ①S…　Ⅱ. ①刘…　②林…　③方…　Ⅲ. ①数据处理软件　Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字(2016)第 208312 号

责任编辑：刘　洋
封面设计：陈国风
责任校对：王荣静
责任印制：沈　露

出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn，http://www.wqbook.com
　　地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
　　社 总 机：010-62770175　　**邮　　购**：010-62786544
　　投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
　　质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京嘉实印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：13　　**字　　数**：228 千字
版　　次：2016 年 9 月第 1 版　　**印　　次**：2016 年 9 月第 1 次印刷
印　　数：1～3000
定　　价：49.00 元

产品编号：071570-01

前　言

自 2012 年回归校园开始电信与互联网大数据分析科研生涯，我与 Hadoop 那头黄色小象就结下了不解之缘。感谢 Google 的论文、Yahoo 的资助、Doug Cutting 无与伦比的聪明才智，以及 Hadoop 开源社区无私奉献的参与者，让成千上万跟我们一样的中小开发者团队拥有了低成本处理大规模数据的能力。HDFS、MapReduce、Pig、Hive、HBase 这些技术组件，帮助我们完成了一个又一个 TB 甚至 PB 级数据集的分析任务。那头可爱的黄色小象，陪伴我度过了一个又一个美好的日子。多么希望这种只用一个技术族就能解决各种大数据处理问题的美好日子能一直持续下去，相信这也是很多开发者梦寐以求的理想国度。然而，梦想终归是梦想。在两年前的某一天，无意中从网络上的一篇技术文章中看到了 Spark 这一新兴技术，文中宣称 Spark 性能和功能均优于 Hadoop。将信将疑的我按照文章中的线索找到了 Spark 官网，下载解压后经过短暂试用，我就被 Spark 的简洁、高效、灵活的特性彻底迷住了。从那时起我就知道，Hadoop，我心目中大数据处理王者技术上的真正挑战者到来了。Spark 以分布式内存对象架构为基础，以 RDD 转换模式为核心，并辅以丰富的 RDD 算子，不仅解决了大数据处理迭代任务的性能问题，还将开发者从简陋的 Map/Reduce 编程模式中解放出来，以更加灵活的方式控制数据的计算过程，并激发无穷的创意。因此，我们的团队逐渐将数据处理技术栈由 Hadoop 转向 Spark。在这个过程中，我们发现目前已有的 Spark 相关书籍大多集中在介绍 Spark 技术的基础原理以及 Spark 相关工具（例如 SparkSQL、SparkR 等）的基本使用方法上。而要学习如何使用 Spark 中提供的丰富算子进行算法设计时，只能以大浪淘沙的方式从网络中零散的资料中寻找参考。因此，我们觉得如果有一本能以丰富示例介绍 Spark 程序和数据挖掘算法设计的书籍，应当能更好地帮助 Spark 开发者提高学习效率，这也就是我们撰写本书的原动力。

基于这一原动力，本书突出以实例的方式介绍和展示 Spark 程序和算法设计的方法。第 1 章以科技史上最为著名的 6 个失败预言引出了大数据时代以及 Hadoop 技术出现的必然性，然后通过 Hadoop 与 Spark 的对比揭示了 Hadoop 的局限性和 Spark 的优势。第 2 章以简洁明了的方式说明了如何以最快的方式搭建一个 Spark 运行环境，并通过 Shell 环境体验 Spark 的强大功能。第 3 章以图文并茂的形式讲解了 Spark 的工作原理、架构与运行机制，并着重介绍了 Spark 的核心 RDD 的变换过程。第 4 章以大量示例代码的形式详细说明了 Spark 丰富的算子，包括创建算子、变换算子、行动算子

和缓存算子。为了帮助读者掌握使用 Spark 设计和实现复杂算法的方法，第 5 章以 10 个常见算法实例展示了 Spark 处理复杂数据处理工作的能力。第 6 章从合理分配资源、控制并行度等 9 个方面介绍了优化 Spark 性能、拓展 Spark 功能的方法。

与市面上大部分 Spark 书籍不同，除原理性文字外，本书还提供了大量的 Spark 代码实例，完成这些代码是一项艰巨的工作。因此，除本书的作者外，我们必须要感谢为文中代码编写和测试作出了巨大贡献的参与者，他们是来自北京邮电大学数据科学中心的研究生梁阳、林澍荣、王蒙、秦超、邱德扬等同学，以及北京浩瀚深度信息技术股份有限公司大数据专家张硕、宋若宁。

由于作者水平有限，加之开源社区的高度活跃性，Spark 技术仍在快速发展中。因此，书中难免会存在不足之处，还请读者见谅并批评指正。意见、建议或交流请发电子邮件至 liujun@ bupt. edu. cn。

北京邮电大学 数据科学中心 刘军

2016 年 8 月

目　录

第 1 章　从 Hadoop 到 Spark ······ 1

1.1　Hadoop——大数据时代的火种 ······ 1

1.1.1　大数据的由来 ······ 1

1.1.2　Google 解决大数据计算问题的方法 ······ 5

1.1.3　Hadoop 的由来与发展 ······ 10

1.2　Hadoop 的局限性 ······ 12

1.2.1　Hadoop 运行机制 ······ 13

1.2.2　Hadoop 的性能问题 ······ 15

1.2.3　针对 Hadoop 的改进 ······ 20

1.3　大数据技术新星——Spark ······ 21

1.3.1　Spark 的出现与发展 ······ 21

1.3.2　Spark 协议族 ······ 24

1.3.3　Spark 的应用及优势 ······ 25

第 2 章　体验 Spark ······ 28

2.1　安装和使用 Spark ······ 28

2.1.1　安装 Spark ······ 28

2.1.2　了解 Spark 目录结构 ······ 31

2.1.3　使用 Spark Shell ······ 32

2.2　编写和运行 Spark 程序 ······ 35

2.2.1　安装 Scala 插件 ······ 35

2.2.2　编写 Spark 程序 ······ 37

2.2.3　运行 Spark 程序 ······ 42

2.3　Spark Web UI ······ 45

2.3.1　访问实时 Web UI ······ 45

2.3.2　从实时 UI 查看作业信息 ······ 46

第 3 章 Spark 原理 …… 50

3.1 Spark 工作原理 …… 50
3.2 Spark 架构及运行机制 …… 54
3.2.1 Spark 系统架构与节点角色 …… 54
3.2.2 Spark 作业执行过程 …… 57
3.2.3 应用初始化 …… 59
3.2.4 构建 RDD 有向无环图 …… 62
3.2.5 RDD 有向无环图拆分 …… 64
3.2.6 Task 调度 …… 68
3.2.7 Task 执行 …… 71

第 4 章 RDD 算子 …… 74

4.1 创建算子 …… 74
4.1.1 基于集合类型数据创建 RDD …… 74
4.1.2 基于外部数据创建 RDD …… 76
4.2 变换算子 …… 80
4.2.1 对 Value 型 RDD 进行变换 …… 80
4.2.2 对 Key/ Value 型 RDD 进行变换 …… 92
4.3 行动算子 …… 108
4.3.1 数据运算类行动算子 …… 108
4.3.2 存储型行动算子 …… 117
4.4 缓存算子 …… 119

第 5 章 Spark 算法设计 …… 123

5.1 过滤 …… 123
5.2 去重计数 …… 125
5.3 相关计数 …… 127
5.4 相关系数 …… 130
5.5 数据联结 …… 135
5.6 Top-K …… 139

5.7　K-means …… 142
5.8　关联规则挖掘 …… 146
5.9　kNN …… 152
5.10　朴素贝叶斯分类 …… 155

第 6 章　善用 Spark …… 161

6.1　合理分配资源 …… 161
6.2　控制并行度 …… 168
6.3　利用持久化 …… 173
6.4　选择恰当的算子 …… 177
6.5　利用共享变量 …… 181
6.5.1　累加器变量 …… 182
6.5.2　广播变量 …… 184
6.6　利用序列化技术 …… 186
6.7　关注数据本地性 …… 188
6.8　内存优化策略 …… 191
6.9　集成外部工具 …… 195

参考文献 …… 198

第1章

从 Hadoop 到 Spark

说起 Spark,就不得不提到可以说是 Spark“前任”的 Hadoop 技术。同为目前最为炙手可热的两个大数据计算框架,Hadoop 与 Spark 经常被放在一起进行比较。受 Google 大数据计算框架启发而产生的 Hadoop 由于出现时间较早,并且由于其大幅降低了编写分布式并行计算程序的难度,同时具备优秀的低成本和可扩展特性,从 2008 年成为 Apache 顶级项目起,Hadoop 用不到 10 年的时间颠覆了历史悠久的大数据处理技术格局,成为当之无愧的大数据处理技术“无冕之王”。然而,由于 Hadoop 的设计重点是解决大数据量情况下的批量运算问题,因此在计算模式多元化(例如迭代和图计算)和实时性要求更高(流式计算和实时计算)的新环境下显得有点“老态龙钟”。于是,强调迭代计算下的性能以及兼容多种计算模式的 Spark 技术应运而生,并在很短的时间内形成了全面取代 Hadoop 之势。为了理清两者的关系以更好地理解 Spark,在本书的开篇我们就从大数据的出现和 Hadoop 的产生说起,通过简要剖析 Hadoop 的原理以说明其局限性,并用一个简单的常用算法实例的运行性能对比,为大家展示 Spark 的强大能力。

1.1 Hadoop——大数据时代的火种

1.1.1 大数据的由来

1965 年 4 月 19 日,时任仙童半导体公司工程师,后来创建英特尔公司的戈登·摩尔在著名的《电子学》杂志(*Electronics Magazine*)发表文章,预言半导体芯片上集成的晶体管和电阻数量将每年增加 1 倍。10 年后,摩尔在 IEEE 国际电子组件大会上将他的预言修正为半导体芯片上集成的晶体管和电阻数量将每两年增加 1 倍,这就是著名的“摩尔定律”[1](如图 1-1 所示)。谁也想不到,这个预言犹如一只看不见的大手推动

着半导体行业在半个世纪里的飞速发展,并见证了以此为基础的 IT 产业的蓬勃发展。然而,不是所有人都能有摩尔这样的洞察力和幸运。在变幻莫测的科技界,更多自信满满的预言者被无情的现实击败。下面,就让我们一起来回顾一下科技史上最为著名的 6 个失败预言,并见证随着这些预言逐一破灭而到来的大数据时代。

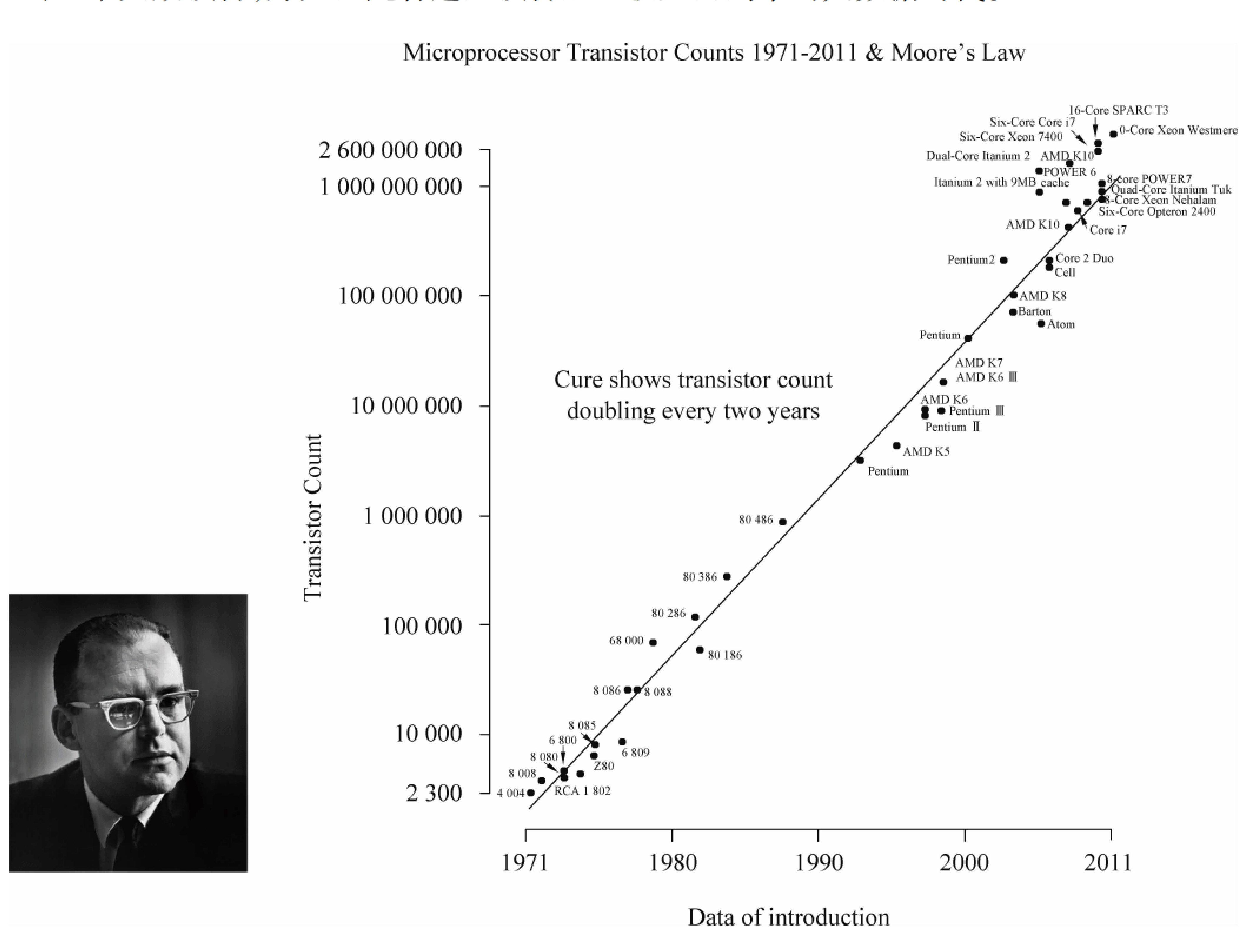


图 1-1 摩尔和摩尔定理

- **最失败预言一:“我找不到普通家庭也需要计算机的理由。”**

——肯 · 奥尔森(Ken Olsen),数字设备公司(DEC)创始人

1977 年,美国数字设备公司(DEC)创始人肯 · 奥尔森认为,普通的家庭是不会花费巨资来购买一台计算机的。他认为普通家庭既承担不起计算机那昂贵的价格,也不需要计算机如此强大的计算能力。当然,后来的故事我们已经知道,随着 Apple、IBM 等推出价廉物美的个人计算机,PC 迅速普及到普通家庭。早在 2011 年在美国进行的一次关于 PC 拥有量的调查就显示,仅拥有一台 PC 的家庭就占到 14% ,而约 60% 的家庭拥有 3 台或者 3 台以上 PC。在中国市场,2014 年全国家庭电脑普及率已超过 50% ,在经济较为发达的城市中,家庭电脑普及率已超过 80% 。很多家庭不仅拥有供家庭娱乐或办公用的台式机,还会购买一些出门旅行或者移动工作时需要的便携式电脑,包

括笔记本、上网本、平板电脑等,甚至还有少数人会拥有自己的服务器。毫无疑问,计算机已成为我们日常生活的一部分。通过计算机,我们每个人都能够以前人所无法想象的速度产生、处理和消费数据。

- **最失败预言二:“很多人预测 1996 年互联网产业将大规模增长。但我的预测是,1996 年互联网产业由于增长过于快速,将像超新星一样爆炸后而走向崩溃。”**

——罗伯特·迈特卡尔夫(Robert Metcalfe),3Com 公司创始人

1995 年,美国公共电视网(Public Broadcasting Service,PBS)推出了一档电视节目,名为《计算机记事》(*Computer Chronicles*)。这个节目介绍了当时还只是少数极客玩具的互联网相关技术、软件和服务。由此,互联网正式开始进入美国公众视野,随之而来的是快速增长的互联网用户并给当时有限的网络带宽带来了极大的压力。在这一背景下,网络设备制造商 3Com 创始人罗伯特·迈特卡尔夫在 *InfoWorld* 发文作出了悲观的预测。他预测 1996 年互联网产业将由于增长过于快速而走向崩溃,并表示如果事实情况证明自己该预测有误,将当众“自食其果”。显然,互联网并没有如他预测的那样走向衰败,反而日益繁荣。因此,在 1999 年举行的国际互联网大会上,迈特卡尔夫在众目睽睽之下,把印有这一预测文字的纸张搅拌到一杯水中,然后一饮而尽。到了 21 世纪的今天,互联网已经渗透到经济社会各领域,给每个人的生活带来了前所未有的改变。Google、Facebook、Youtube、Twitter、QQ、微博、微信等丰富多彩的互联网服务,为每个人打开了一扇从网络了解世界、向世界分享自我的窗口,在互联网中自由平等地交换数据与信息。

- **最失败预言三:“全球垃圾邮件问题将在今后两年内得到解决。”**

——比尔·盖茨(Bill Gates),微软创始人

随着 20 世纪 90 年代互联网的出现和发展,由于互联网及相关服务的高度开放性和交互性,任何一个网站或个人都能生产和发布信息,这为所有信息的传播开辟了一个几乎不受限制的空间。在此之前,全球的数据量基本是每 20 个月增加 1 倍。而在互联网出现之后,数据量则呈现出几何级数增长的趋势。然而,在暴涨的数据中,并不完全是对用户有益,其中还充斥着大量垃圾数据,垃圾邮件就是其中最臭名昭著的一类。比尔·盖茨,微软帝国的缔造者,毫无疑问是一位“技术控”,他坚信随着技术的发展垃圾邮件必然会得到有效控制,并且这一目标将很快实现,互联网会因此变得更加安全。因此,2004 年 11 月在马德里举行的一次互联网大会上,比尔·盖茨表示:“目前垃圾邮件已成为全球非常严重的安全问题,业界还没有找到有效遏制措施,但我们

希望这一现象在两年之内得到控制。”遗憾的是，现实比他预言的要残酷得多，直到今天垃圾邮件仍然是困扰全球互联网用户的顽疾。就在笔者编写本段文字的时候，在屏幕的右上角就在弹出邮件软件收到的垃圾邮件提示信息。

- **最失败预言四：“电视节目的流行时间不会超过半年，公众每晚会面对着一个小盒子，他们将对此感到厌烦。”**

——达里尔·扎努克（Darryl Zanuck），20世纪福克斯公司高管

在20世纪中期，电视机虽然已经开始走入普通家庭，开启了影音娱乐时代。但早期的电视机可以说只是一个简陋的黑白图像接收器，在这样的设备上观看娱乐节目的体验可想而知。因此，虽然像20世纪福克斯这样的传统电影制作和发行公司感觉到了电视所带来的挑战，但他们仍然认为这种简陋的娱乐形式很快会随着人们对电视机新鲜劲的过去而被淘汰，这也是为什么达里尔·扎努克会预言电视节目的寿命会非常短暂的原因。然而，扎努克严重低估了人们对即时影音媒介的强烈需求以及技术所带来的变化。随着彩色电视、无线技术和光通信技术的发展，电视节目从模拟信号发展为数字信号。同时，电视节目的传输媒介已经不仅仅局限在无线电波和有线电视线缆，人们可以通过PC、手机、平板电脑等多种终端收看电视节目。收看电视节目已经不仅仅是为了人们每晚的休闲娱乐，更是人们认识世界、了解世界的途径。人们对富媒体的需求不但没有减退，反而与日俱增。同时，人们对电视节目的视觉效果也提出了更高的要求，从标清到高清再到超高清，人们不断提出着对视觉效果的更高要求并得到满足。清晰度的不断提升，意味着在画面大小不变的情况下，需要有更多的像素点，这也带来了在网络中传输数据的快速膨胀。

- **最失败预言五：“苹果已死。”**

——纳桑·梅沃尔德（Nathan Myhrvold），微软前首席技术官

1976年，21岁的史蒂夫·乔布斯和他的小伙伴们研制出了世界上第一台个人电脑 Apple-I。虽然 Apple-I 的计算能力与现在的个人电脑相比相差甚远，但它用事实证明了个人电脑完全可以进入普通家庭。1984年，全新的苹果电脑 Macintosh 诞生，这是世界上第一款采用交互式图形处理界面的个人电脑产品。Macintosh 的成功帮助苹果公司达到第一个巅峰，其1985年的股票市值高达20亿美元。但由于各种原因，苹果公司自此开始从巅峰滑落，将个人电脑市场的统治地位拱手让给微软。正是在这一背景下，微软前首席技术官纳桑·梅沃尔德推断苹果公司不久就会倒闭。然而，梅沃尔德和其他人显然都低估了乔布斯的能量。1997年，乔布斯重回苹果公司执掌大权，强

力推动苹果公司回归精品创新战略,接连推出了 iPod、iPhone、iPad 等一系列让消费者眼前一亮的产品,并带领苹果公司重返巅峰。苹果公司推出的一个又一个智能移动设备不仅帮助苹果公司成为市场霸主,也极大地推动了移动互联网的发展。在 iPhone、安卓手机的帮助下,移动互联网再次改变了我们的生活方式。人们可以随时随地分享文字、图片、音频和视频,这些来自个人的多媒体数据在互联网上再次掀起了一股数字洪流。

- **最失败预言六:"我觉得全球市场大概只需要 5 台计算机。"**

——马斯·沃森(Thomas Watson),IBM 前总裁

当沃森作出上述预言时,全球计算机产业正处于初创阶段。个人电脑还没有如此普及,互联网企业还没有大规模兴起,更没有出现如今这么多的智能设备。1GB 的数据在当时就已经是一个"大"数据了。因此,沃森认为 5 台计算机(当然他指的是 IBM 的大型机)就完全可以满足全世界所有的数据计算需求了。然而,现实世界数据量和计算需求的增长速度完全超出了沃森的想象。我们来看一下目前一些大型互联网企业需要处理的数据量级:①百度 > 200PB;②Facebook > 100PB;③Yahoo > 100PB;④淘宝 > 15PB。不仅是互联网,其他各个行业的数据均呈现出爆炸式的增长。大数据已经渗透到当今社会的各个领域中。超大规模的数据集已经从拍字节(PB)发展到艾字节(EB),再到泽字节(ZB),而且数据的增加速度还在不断地加快。如此大的数据量别说是计算,就是存储,5 台计算机也存不下。

以上的预言之所以失败,皆是因为他们低估了科技发展和人们需求增长的速度。现如今,我们正处在一个科技高速发展的时代。随着计算机技术和互联网技术的高速发展以及信息的数字化,使得科技水平日新月异,新科技、新应用层出不穷。同时也带来一个新的挑战:我们需要面对的数据量越来越大。我们每天使用的个人电脑以及各种智能设备,我们每天收看的各种视频节目,我们每天享用的各种便捷的互联网业务都在不断产生新的数据。科技的高速发展用事实认证了 6 个预言的失败,同时也将我们带入了一个"大数据时代"。

1.1.2 Google 解决大数据计算问题的方法

随着上面 6 个预言的破灭,我们正式迎来了"大数据时代"。但是在很长一段时间里,人们面对这爆炸的数据显得有点无所适从,如何从海量的数据中找到自己想要的信息成为困扰很多人的难题。于是,Google 利用这一难得的历史机遇成长为全球最大的搜索引擎服务提供商。我们都知道,Google 并不是第一个做搜索引擎的,在它之前

还有 Yahoo、AltaVista 等搜索引擎。那么,Google 是如何战胜这些强大的竞争对手脱颖而出的呢?下面我们就一起来揭开这谜题,你将看到这个谜题的答案是如何为我们带来大数据时代的无冕之王 Hadoop 的。

提到 Google 搜索引擎,通常我们首先想到的是 Google 的核心算法——PageRank 算法[2]。PageRank 算法是 Google 创始人佩奇(Page)的博士论文题目,有意思的是,PageRank 中的 Page 并不是网页(Web page),而是佩奇(Page)。佩奇在寻找博士论文题目时,被互联网的数学特性所深深吸引。他发现互联网中的每个网页都可以看作是一个节点,而网页上的链接就是这些节点间的联系,它们一起构成了一个经典的数据结构——图(graph)结构。佩奇参考了基于论文被引用率评估某篇论文重要性的思想,发现如果将网页 A 指向网页 B 的一个链接视为网页 B 的重要性来源,同时综合考虑网页 A 自身的重要性,就可以准确地得到网页 B 的重要性度量值,这就是 PageRank 算法的基本思想。在这里我们不具体探讨 PageRank 算法,感兴趣的读者可以自行探索。我们只需要知道利用 PageRank 算法可以得到每个网页的重要性,从而帮助使用者找到与他搜索请求最匹配的网页。但是,要达到这一目标,还必须解决 3 个关键问题:①如何高效存储和管理抓取获得的数以亿计的网页信息?②如何快速计算数以亿计网页的 PageRank 值?③如何在收到用户请求后快速返回匹配结果?同时,在当年也只是一个普通初创互联网公司的 Google,要完成以上任务,还必须满足以下两个约束条件:①用较低的硬件成本以确保公司的可赢利性;②容易扩展以适应互联网数据的快速膨胀。这看起来似乎是个不可能完成的任务,然而 Google 的计算机天才们在强大的压力下研发出了解决这一大数据处理问题的三大神器:具备海量数据存储和访问的分布式文件系统 GFS[3]、简洁高效的并行计算编程模型 MapReduce[4]、支持海量结构化数据管理的 BigTable[5]。这 3 项技术不仅为 Google 采用大量廉价计算机实现包括搜索业务在内的海量数据处理能力提供了技术基础,还直接导致了目前被广泛应用的 Hadoop 大数据计算架构的产生。由于篇幅的限制,下面我们只简要地了解下与本书主题 Spark 相关的两个基础技术 GFS 和 MapReduce。

- **分布式文件系统 GFS**

Google 的系统中存储了大量通过互联网抓取的各种网页信息,这些信息需要存储在文件中进行管理。由于其搜索业务需要管理的海量数据特征和操作特性与传统的分布式文件系统有较大不同,因此,在 Google 发展早期两位创始人编写了自有的文件管理系统 BigFiles,并在此基础上发展出了 Google File System(GFS)。GFS 是一个可扩展的分布式文件系统,可用于大型分布式的海量数据文件的管理。GFS 的设计思想不

同于传统的分布式文件系统,其针对大规模数据处理和 Google 应用特征进行了特殊设计,依靠容错机制运行于低成本的普通硬件上,为大量并发用户提供总体性能较高的文件管理服务。GFS 设计的出发点与传统分布式文件系统的区别主要有以下几点。

(1) 系统构建在由大量低成本的普通计算机组件构成的底层硬件资源上,系统中不可避免地会出现较多的系统故障。因此,在基本文件管理功能之外,还需要随时监测故障的发生,并引入容错机制避免故障对整体计算过程的影响。

(2) 系统要管理的文件主要由较大尺寸的文件构成,通常数量在百万数量级,文件大小在 100MB 以上,且可能存在较多的大小为吉比特级别的文件,对这些大文件的管理需要进行优化,以提高效率。同时,系统中也会存在小文件,同样需要进行支持,但无需进行额外优化。

(3) 系统中的主要负载为两类文件操作,大数据量流式读取和小数据量随机读取。在大数据量流式读取操作中,每个操作需要读取数百千比特或 1 兆比特以上的数据。由同一客户端发起的读取操作通常会读取一个文件的若干连续区域。小数据量随机读取则通常在随机的位置读取几千比特的数据。对性能要求较高的应用通常会对小数据量随机读取进行优化,会对这些操作进行排序并进行批量处理,以确保磁盘在访问文件时能连续读取而非反复寻址。

(4) 系统中的负载还存在很多大数据量的对文件的附加写入数据的操作,这些操作的数据量与读操作类似。文件在写入后很少会被修改。对文件的小数据量的随机写操作需要被系统支持,但由于极少发生因此不需要额外关注。

(5) 系统需要实现完整定义的语义操作以支持多个终端对一个文件同时进行附加数据的操作。这是因为系统中的文件经常被多个队列使用或者进行多方合并操作,因此同一个文件可能会被成百上千个终端进行附加数据操作。

(6) 系统对底层网络可持续的高带宽数据传输的要求要大于对传输时延的要求,这是因为系统所支持的应用通常需要快速的大数据量处理,而很少对单个读写操作的响应时延有特殊的要求。

基于以上特性的需求,Google 设计的 GFS 系统架构参见图 1-2 所示。

GFS 架构中存在 3 类节点:客户端、主控节点、块服务器。客户端(Client)是 GFS 提供给上层应用使用的一组接口库,上层应用通过调用接口库中的接口实现 GFS 系统中的文件管理。主控节点(Master)是 GFS 系统的管理节点,在逻辑上,整个系统中只存在一个主控节点。主控节点中保存了系统中所有文件的元数据,并负责块服务器的调配。块服务器(Chunk Server)中存储了具体的数据文件块(Chunk),系统中有多个

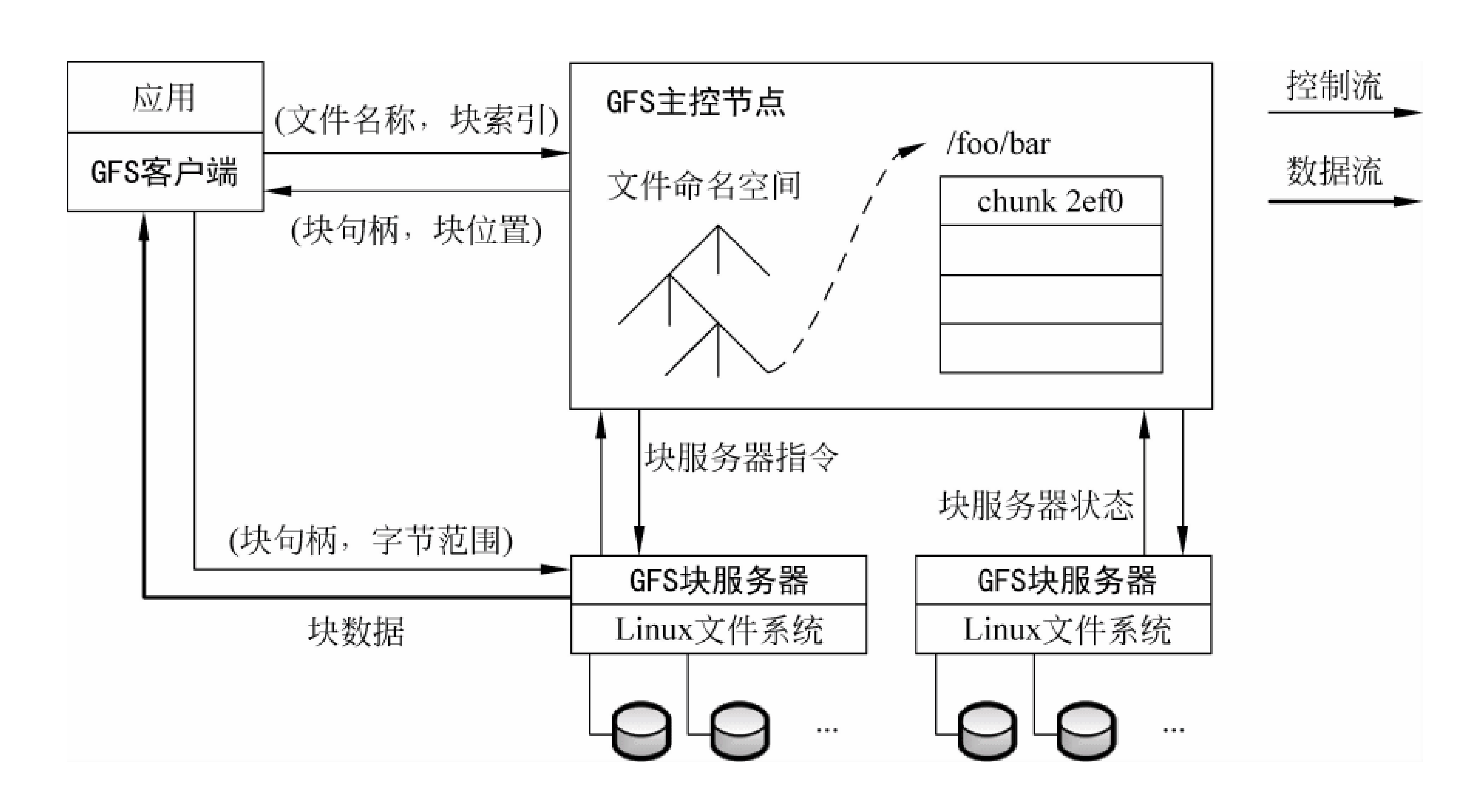


图 1-2 GFS 系统架构

块服务器。文件按照固定大小的块进行存储，默认的文件块大小为 64MB，每个文件块有一个唯一的索引号。当一个应用要访问文件时，客户端将应用提供的文件名称和字节偏移通过固定文件块大小进行计算后获得块索引，然后将文件名称和块索引发送给主控节点，主控节点将相应的用于访问文件块的块句柄和文件块所在的块服务器位置返回给客户端。客户端将这些信息进行缓存，并访问离自己距离最近的块服务器获得块数据。块服务器信息的缓存减少了客户端和主控节点间的数据交互，提高了系统运行效率。

- **分布式并行计算编程模型 MapReduce**

MapReduce 编程思想的起源来自于 1956 年由图灵奖获得者 John McCarthy 提出的 Lisp 语言中的 MapReduce 功能。Lisp 语言的 MapReduce 操作模式是一种标准的函数式编程模式（其他的编程模式还有我们熟悉的面向过程编程模式、面向对象编程模式等），这种编程模式将计算过程转化为一系列数学函数的计算。由于这种模式中不引入变量，且不具有状态性，因此非常适合并行计算环境。在 1995 年，John Darlington 等人将这种思想以 Map、Fold 的形式引入到他们提出的并行编程语言 SCL 中。

在借鉴以上编程思想的基础上，Google 实现了自己的 MapReduce 并行编程模型。与以上基础的编程函数不同，Google 的 MapReduce 不仅是一个简单而强大的函数接口，而且包含了一系列并行处理、容错处理、本地化运算和负载均衡等技术的软件实现，提供了一个完整的大尺度并行运算编程环境。简单来说，Map 操作就是对输入的一个数据列表中的每个元素进行一个指定的操作，而 Reduce 操作就是将 Map 输出的新数据列表以某种方式进行合并操作。Google 的 MapReduce 运行原理和实现机制参

见图 1-3。

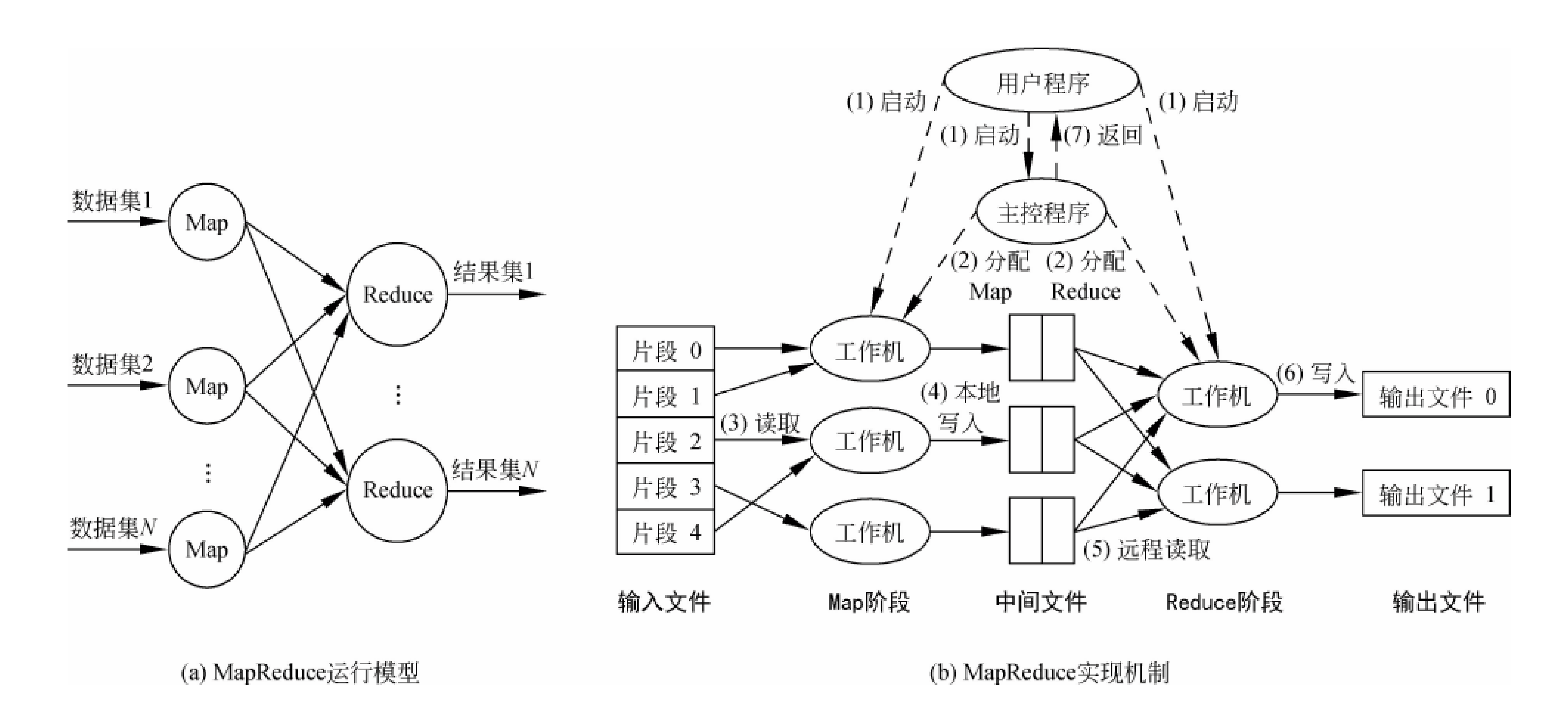


图 1-3 MapReduce 原理和机制

简单来说,Map 操作就是对输入的一个数据列表中的每个元素进行一个指定的操作,而 Reduce 操作就是将 Map 输出的新数据列表以某种方式进行合并操作。在 Google 的实现机制中,主要有两类特殊的工作节点,一类是工作机(Worker),用于执行 Map 或 Reduce 任务,一类是主控程序(Master),用于将 Map 和 Reduce 分配到合适的工作机上。图 1-3(b)描述了用户程序、主控程序和工作机协同工作完成整个 MapReduce 工作的流程。

(1)用户程序中的 MapReduce 库将输入的数据文件分为 M 个片段,每个片段的大小在 16～64MB,然后在计算机集群上启动很多程序副本。

(2)其中的一个程序副本被指定为主控程序,其余的为工作机。主控程序指定 M 个空闲的工作机运行 Map 任务,R 个空闲的工作机运行 Reduce 任务。

(3)被指定的 Map 工作机从对应的输入文件片段中读取需要处理的数据集,并进行处理获得中间结果。

(4)Map 工作机产生的中间结果先被缓存在内存中,并定期写入每个 Map 工作机的本地硬盘。写入本地硬盘的中间结果通过分区函数被分为 R 个分区,并且中间结果在本地硬盘的位置信息会发送到主控程序,由主控程序通知 Reduce 工作机。

(5)当 Reduce 工作机收到主控程序发来的中间结果位置信息后,通过远程处理请求将位于 Map 工作机本地硬盘的数据读取到 Reduce 工作机中,以准备进行相应的处理。

(6)Reduce 工作机在读取到所需要的全部中间数据后,对数据进行排序,并按照指定的 Reduce 函数进行处理,结果被输出到一个最终的输出文件中。

（7）在所有的 Map 和 Reduce 任务执行完成后，主控程序将激活用户程序，用户程序的执行回到 MapReduce 请求的发生点。

2003 年至 2006 年期间，Google 将其三大核心技术 MapReduce、GFS、BigTable 以学术论文的形式进行了公开发表，还披露了与这三大技术相关的其他几项构成其大数据计算平台的关键技术，例如分布式锁服务 Chubby、海量数据分析语言 Sawzall 等。这一技术簇群的公布，迅速在海量数据处理业界引起了巨大反响，并直接导致了目前被广泛采用的开源架构 Hadoop 的诞生。

1.1.3 Hadoop 的由来与发展

当 2003 年 Google 第一篇关于云计算核心技术 GFS 的论文发表时，Apache 开源项目 Nutch 搜索引擎的开发者 Doug Cutting 等人正面临着如何将其架构扩展到可以处理数十亿网页的规模的难题。在了解了 GFS 系统后，他们敏锐地意识到，这样的技术架构可以帮助他们解决存储 Nutch 抓取网页和建立索引过程中产生的大量文件的问题，并提高管理这些存储节点的效率。因此，在参考 GFS 技术的基础上，他们在 2004 年编写了一个开放源码的类似系统 NDFS 分布式文件系统（Nutch Distributed File System）[6]。

同样在 2004 年，Google 公开发表了阐述其另一核心技术 MapReduce 的论文，让业界第一次真切感受到了 MapReduce 编程模型在解决大型分布式并行计算问题上的巨大威力和实用性。很快 Nutch 团队就将 MapReduce 技术应用于他们的项目，在 2005 年将 Nutch 的主要算法都移植到基于 MapReduce 和 NDFS 的框架下运行。

在完成了 MapReduce 和 NDFS 的开源实现后，Nutch 项目的两名兼职开发人员为 Nutch 搭建了一个包含 20 个计算节点的平台，验证了这两个开源组件在解决搜索数百万网页问题情况下的有效性。但是，在面对将这两项技术拓展到可以面对数十亿级的网页的工作时，他们面临了绝大的资源压力。幸运的是，雅虎公司也发现了这两项技术的巨大潜力，将 Nutch 的开发者之一 Doug Cutting 招入公司，并建立了一个专门的团队提供支持。在这样的条件下，Nutch 项目的分布式运算部分被单独剥离出来，成为了 Apache 的一个单独子项目 Hadoop。有意思的是，Hadoop 这一项目名称来源于创立者 Doug Cutting 儿子的一个玩具，一头黄色的大象，并没有什么实际的含义。这一简短易读的命名特点影响了 Hadoop 后续子项目的命名，例如 Pig、Hive 等。

Hadoop 项目的目标是建立一个能够对海量数据进行可靠的分布式处理的可扩展开源软件框架。Hadoop 面向的应用环境是大量低成本计算机构成的分布式运算环境，

因此,它假设计算节点和存储节点会经常发生故障,为此设计了数据副本机制,确保能够针对在出现故障节点的情况下重新分配任务。同时,Hadoop 以并行的方式工作,通过并行处理加快处理速度,具有高效的处理能力。从设计之初,Hadoop 就为支持可能面对的 PB 级海量数据环境进行了特殊的设计,具有优秀的可扩展性。可靠、高效、可扩展这三大特性,加上 Hadoop 开源免费的特性,使 Hadoop 技术得到了迅猛发展,并在 2008 年成为 Apache 的顶级项目。

Hadoop 项目对应于 Google 云计算核心技术 GFS、MapReduce、BigTable 实现了自己的三大核心子项目:HDFS、MapReduce 和 HBase。但在实际应用环境中,大数据资源的来源多种多样,各不相同。为了处理特殊的需求,目前 Hadoop 项目已经形成了一个生态圈,包括多个应用工具。下面我们将与 Hadoop 体系相关的由 Apache 开源组织支持的主要组件及相关项目整理如下。

- Hadoop Common:从其名称就可以看出来,Hadoop Common 项目是为 Hadoop 整体架构提供基础支撑性功能,主要包括了文件系统(FileSystem)、远程过程调用协议(RPC)和数据串行化库(Serialization Libraries)。资料地址:https://hadoop.apache.org。
- HDFS:HDFS 是运行在由廉价计算机组成的大规模集群上的分布式文件系统,具有低成本、高可靠性、高吞吐量的特点,由早期的 NDFS 演化而来。资料地址:https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html。
- MapReduce:MapReduce 是一个分布式的数据处理模式和执行环境。用于在大规模计算机集群上编写对海量数据进行快速处理的并行化程序。资料地址:https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html。
- YARN:YARN 是一个集群资源管理调度的框架,用以提高分布式的集群环境下的资源利用率。资料地址:http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html。
- Ambari:Ambari 是一个用于安装、管理和监控 Hadoop 集群的 Web 界面工具。目前已支持包括 MapReduce、HDFS、HBase 在内的几乎所有 Hadoop 组件的管理。项目地址:http://incubator.apache.org/ambari。
- Avro:Avro 是一个数据序列化系统,由 Hadoop 创始人 Doug Cutting 牵头开发,主要提供的功能包括:①丰富的数据结构类型;②一个快速可压缩的二进制格式;③提供一个可以存储持久数据的文件容器;④远程过程调用;⑤可以方

便地和动态语言结合。项目地址：http:// avro. apache. org。

- Cassandra：Cassandra 是一个大型的面向列的分布式数据库，具有弹性可扩展、高可用性、容错能力强等特性。项目地址：http:// cassandra. apache. org。
- HBase：HBase 是一个分布式的、面向列的开源数据库，不同于一般的关系数据库，它是一个适合于非结构化海量数据存储的数据库。项目地址：http:// hbase. apache. org。
- Hive：Hive 是一个分布式的数据仓库。Hive 管理的数据全部存储在 HDFS 上，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为可执行的 MapReduce 任务进行运行。项目地址：http:// hive. apache. org。
- Mahout：Mahout 是一个可扩展的机器学习和数据挖掘的算法库。Mahout 现在已经包含了聚类、分类、推荐引擎（协同过滤）和频繁项集挖掘等广泛使用的数据挖掘方法。项目地址：http:// mahout. apache. org。
- Oozie：Oozie 是一个 Hadoop 工作流调度系统，它可以把多个 Map/ Reduce 作业组合到一个逻辑工作单元中，从而完成更大型的任务。项目地址：http:// oozie. apache. org。
- Pig：Pig 是一个处理大数据的数据流式语言和执行环境。运行在 HDFS 和 MapReduce 集群之上。Pig 的特点是其结构设计支持真正的并行化处理，因此适合应用于海量数据处理环境。项目地址：http:// pig. apache. org。
- Sqoop：Sqoop 是一款用于在 HDFS 与传统关系数据库间进行数据交换的工具。可以用于将传统数据库（例如 Mysql、Oracle）中的数据导入 HDFS 或 MapReduce，并将处理后的结果导出到传统数据库中。项目地址：http:// sqoop. apache. org。
- ZooKeeper：ZooKeeper 是一个分布式应用程序协调服务器，用于维护 Hadoop 集群的配置信息、命名信息等，并提供分布式锁同步功能和群组管理功能。项目地址：http:// zookeeper. apache. org。

1.2　Hadoop 的局限性

自 2008 年成为 Apache 顶级项目后，经过 7 年的发展，Hadoop 已经成为大数据处理技术事实上的标配，并在众多应用领域证明了效率和成本的优势，这是一项非常了不起的成就。然而，正如硬币都有正反两面一样，由于过度追求解决批量处理问题，

Hadoop 技术也在面对需要迭代计算或流式处理的应用场景时存在很大的局限性。要了解这一局限形成的原因，我们还需要从 Hadoop 技术的两个核心组件 HDFS 和 MapReduce 说起。

1.2.1 Hadoop 运行机制

图 1-4 展示了一个典型的 Hadoop 部署环境图及逻辑组件之间的交互，我们将结合此图对 Hadoop 的主要逻辑组件进行说明，并为大家建立一个简明的 Hadoop 原理和运行机制全景图。由于篇幅所限，我们在这里并不对 HDFS 和 MapReduce 的技术细节进行说明，感兴趣的读者可以通过本书作者编写的《Hadoop 大数据处理》一书进行了解。Hadoop 是通过 HDFS 和 MapReduce 的两大逻辑组件相互配合完成用户提交的海量数据处理请求，它们的功能组件如下（以较易理解的 Hadoop 版本 1.x 的架构为例）：

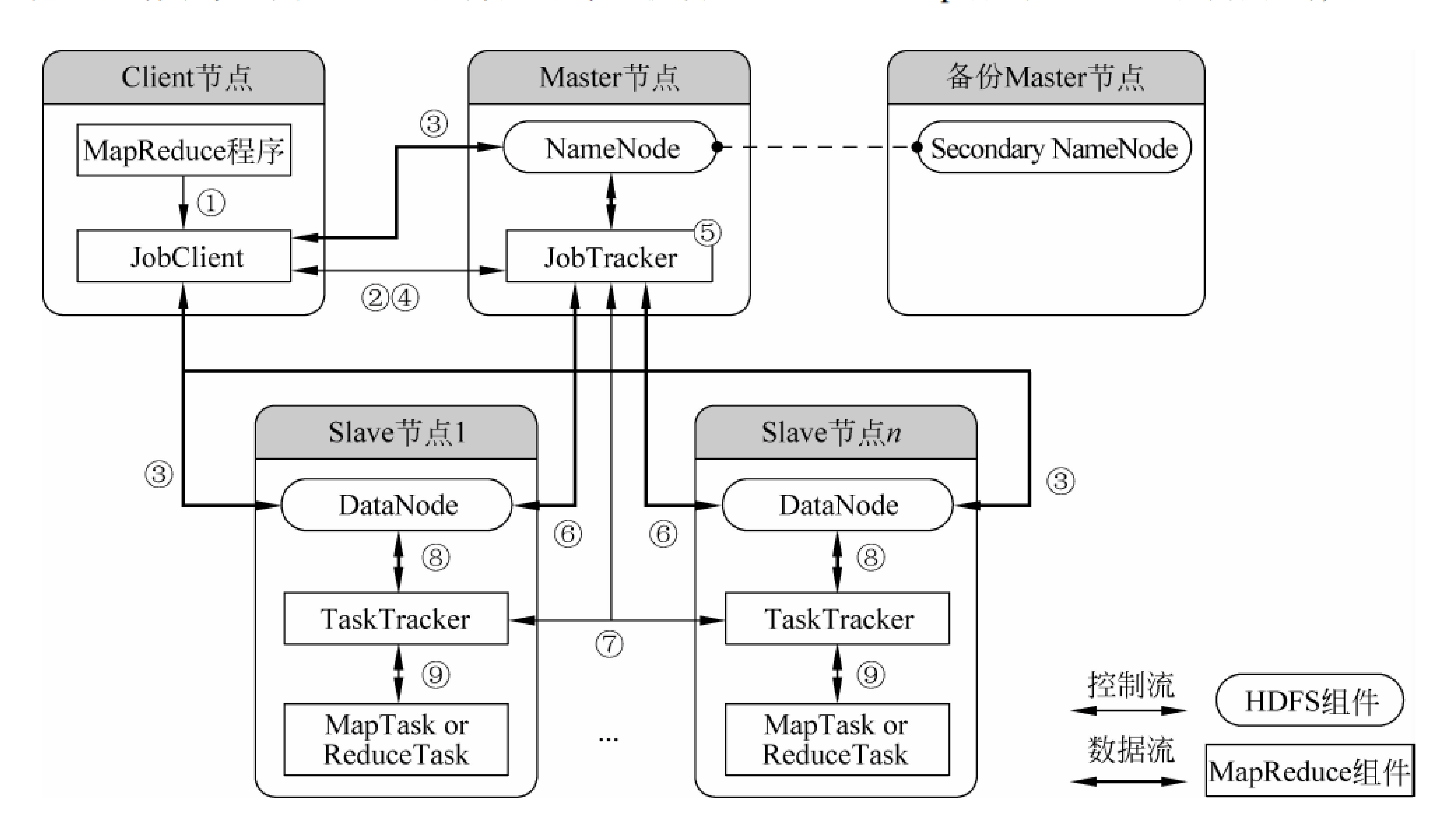


图 1-4 Hadoop 部署及逻辑组件关系图

- **HDFS 组件**

√ NameNode：NameNode 是 HDFS 系统中的管理者，它负责管理文件系统的命名空间，维护文件系统的文件树及所有的文件和目录的元数据。这些信息存储在 NameNode 维护的两个本地磁盘文件：命名空间镜像文件和编辑日志文件。同时，NameNode 中还保存了每个文件与数据块所在的 DataNode 的对应关系，这些信息将被用于其他功能组件查找所需文件资源的数据服务器。

√ DataNode：DataNode 是 HDFS 文件系统中保持数据的节点。HDFS 中的文件通常被分割为多个数据块，以冗余备份的形式存储在多个 DataNode 中。

DataNode 定期向 NameNode 报告其存储的数据块列表，以备使用者通过直接访问 DataNode 获得相应的数据。

- **MapReduce 组件**

√ JobClient：JobClient 是基于 MapReduce 接口库编写的客户端程序，负责提交 MapReduce 作业。

√ JobTracker：JobTracker 是应用与 MapReduce 模块之间的控制协调者，它负责协调 MapReduce 作业的执行。当一个 MapReduce 作业提交到集群中，JobTrack 负责确定后续执行计划，包括需要处理哪些文件、分配任务的 Map 和 Reduce 执行节点、监控任务的执行、重新分配失败的任务等。每个 Hadoop 集群中只有一个 JobTracker。

√ TaskTracker：TaskTracker 负责执行由 JobTracker 分配的任务，每个 TaskTracker 可以启动一个或多个 Map 或 Reduce 任务。同时，TaskTracker 与 JobTracker 间通过心跳（HeartBeat）机制保持通信，以维护整个集群的运行状态。

√ MapTask，ReduceTask：MapTask 和 ReduceTask 是由 TaskTracker 启动的负责具体执行 Map 任务和 Reduce 任务的程序。

以上这些组件协同工作执行一个分布式并行数据技术任务的流程如下（编号与图中的流程编号对应）。

① MapReduce 程序启动一个 JobClient 实例以开启整个 MapReduce 作业（Job）。

② JobClient 通过 getNewJobId（）接口向 JobTracker 发出请求，以获得一个新的作业 ID。

③ JobClient 根据作业请求指定的输入文件计算数据块的划分，并将完成作业需要的资源，包括 JAR 文件、配置文件、数据块，存放到 HDFS 中属于 JobTracker 的以作业 ID 命名的目录下，一些文件（例如 JAR 文件）可能会以冗余备份的形式存放在多个节点上。

④ 完成上述准备工作后，JobClient 通过调用 JobTracker 的 submitJob（）接口提交此作业。

⑤ JobTracker 将提交的作业放入一个作业队列中等待进行作业调度以完成作业初始化工作。作业初始化主要是创建一个代表此作业的运行对象，作业运行对象中封装了作业包含的任务和任务运行状态记录信息用于后续跟踪相关任务的状态和执行进度。

⑥ JobTracker 还需要从 HDFS 文件系统中取出 JobClient 放好的输入数据，并根据

输入数据创建对应数量的 Map 任务。同时,根据 JobConf 配置文件中定义的数量生成 Reduce 任务。

⑦ 在 TaskTracker 和 JobTracker 间通过心跳机制维持通信,TaskTracker 发送的心跳消息中包含了当前是否可执行新任务的信息,根据这个信息,JobTracker 将 Map 任务和 Reduce 任务分配到空闲的 TaskTracker 节点。

⑧ 被分配了任务的 TaskTracker 从 HDFS 文件系统中取出所需的文件,包括 JAR 程序文件和任务对应的数据文件,并存入本地磁盘,并启动一个 TaskRunner 程序实例准备运行任务。

⑨ TaskRunner 在一个新的 Java 虚拟机中根据任务类别创建出 MapTask 或 ReduceTask 进行运算。在新的 Java 虚拟机中运行 MapTask 和 ReduceTask 的原因是避免这些任务的运行异常影响 TaskTracker 的正常运行。MapTask 和 ReduceTask 会定时与 TaskRunner 进行通信报告进度,直到任务完成。

1.2.2　Hadoop 的性能问题

数据科学家在面对大规模数据分析时,经常需要面对两类问题。

(1) 数据缓存: 在应用数据挖掘算法前,数据往往需要进行预处理操作,对数据中一部分不符合要求的数据进行不断的清洗过滤。而这些清洗工作又不是可以用简单的线性操作完成的。同时,算法计算过程中的中间结果也需要保留,以便后续操作使用。

(2) 算法迭代: 数据科学家需要应用复杂的数据挖掘算法对数据进行分析,而这些算法往往需要复杂的运算逻辑和反复的迭代过程,以达到求解最优解的目的。例如 K-means 算法、梯度下降算法等。

由于 Hadoop 天生的设计缺陷,在处理以上两个问题时显得有点力不从心。为了理解其原因,我们先从 Hadoop 的核心计算模式 MapReduce 说起。MapReduce 计算模式将数据的处理过程分为两个阶段: Map 和 Reduce。在 Map 阶段,原始数据被输入 mapper 进行过滤和转换,获得的中间数据在 Reduce 阶段作为 reducer 的输入,经过 reducer 的聚合处理,获得最终处理结果。其过程可以用图 1-5 描述。

为了适应多样化的数据环境,MapReduce 中采用关键字/ 值数据对(Key-Value Pair)作为基础数据单元。关键字和值可以是简单的基本数据类型,例如整数、浮点数、字符串、二进制字节,也可以是复杂的数据结构,例如列表、数组、自定义结构。Map 阶段和 Reduce 阶段都将关键字/ 值作为输入和输出,其公式表达如下(<… >代表关键

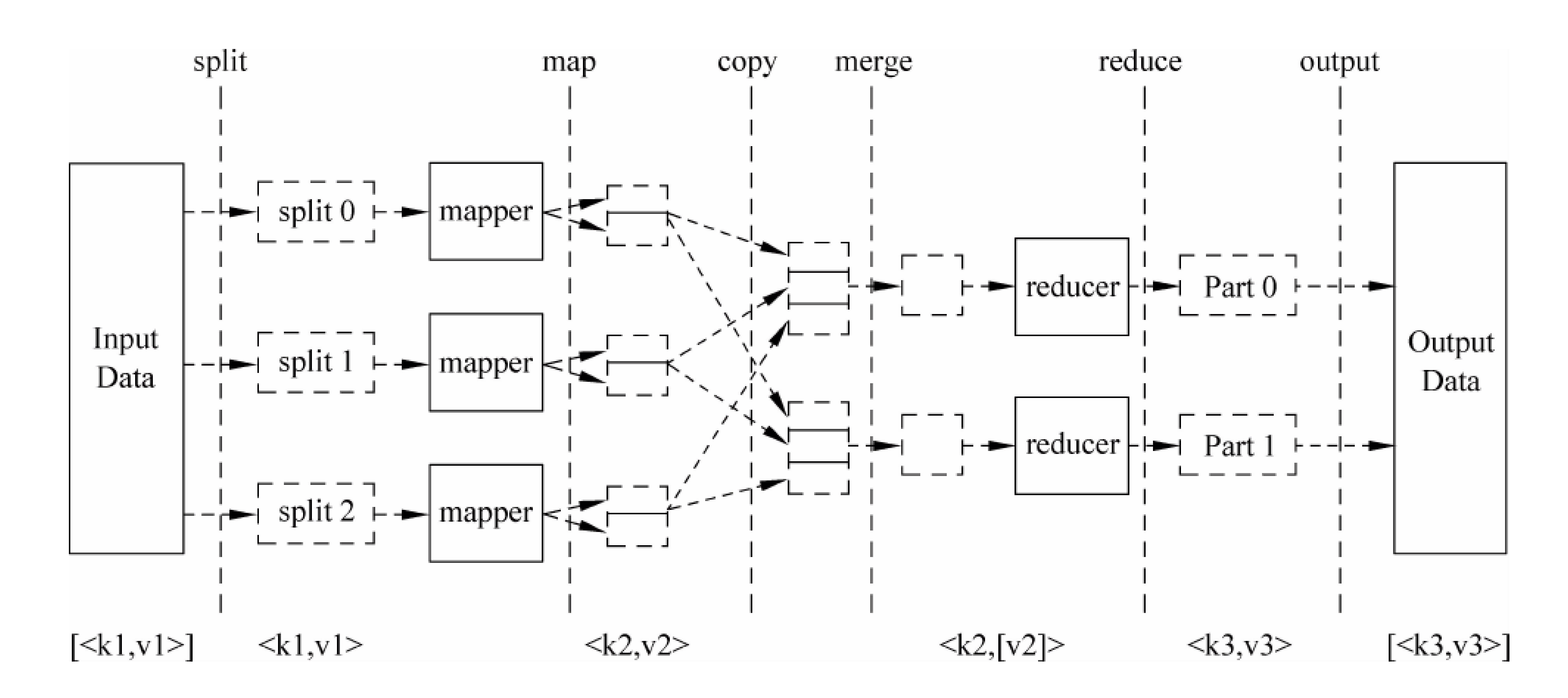


图 1-5 MapReduce 处理过程图

字/值数据对,[…]代表列表):

Map: <k1, v1> → [<k2, v2>]
Reduce: <k2, [v2]> → [<k3, v3>]

结合上图和公式,MapReduce 基本处理过程表达如下。

(1) MapReduce 应用将一个[<k1, v1>]列表作为参数传递给 MapReduce 处理模块,例如[<String fileName, String fileContent>]。MapReduce 处理模块将这个列表拆分为单独的<k1, v2>数据对,分发给对应的 mapper 进行处理。

(2) mapper 根据函数定义的处理过程,对<k1, v1>进行处理,生成<k2, v2>列表。由于数据处理过程是无序的,因此,每个<k2, v2>数值对的生成过程必须是自包含的,即不与其他数值对的生产过程相关。所有 mapper 的处理结果合在一起构成了一个大的<k2, v2>列表,这个列表中关键字相同的数据对被合并为一个新的关键字/值数据对<k2, [v2]>,即 k2 和一系列的 v2。reducer 按照函数定义的处理过程,对这些新的数据对进行处理,获得最终的处理结果,并以[(k3, v3)]列表的形式输出。

通过如上对 MapReduce 的介绍可以看出,MapReduce 工作模型非常简单,它只有两个基本操作:Map、Reduce。这种简化是为了降低编写分布式并行计算程序的复杂度。但是,这种简化也带来了新的问题,就是在进行复杂算法设计和非线性的数据处理时,只能通过 Map + Redcue 的叠加来实现。一个算法通常需要多个 Map + Redcue 的过程才能实现,而每一个 Map + Redcue 的过程,Hadoop 都要单独启动一个 Job 来实现。每一个 Job 的启动都增加了整个算法的开销。同时,MapReduce 计算模式在进行多个 Job 时,必须依赖 Hadoop 的另一项核心技术 HDFS 交换数据。HDFS 最初的设计思想是数据"一次写入、多次读写"。HDFS 中的一个数据块会被复制分发到不同的存储节

点中，使用磁盘作为中间过程交互数据的媒介。多个 MapReduce Job 在衔接时，都需要将数据写入磁盘，再读出，因此，在进行反复迭代的计算时会增加大量的网络开销和磁盘 IO 开销。

为了直观展示 Hadoop 的局限性，我们用一个常见的数据挖掘算法 K-means[7] 为实例来说明。K-means 算法是最为经典的聚类方法，被誉为数据挖掘十大经典算法之一，在数据挖掘领域有着广泛的应用。K-means 算法要解决的问题如图 1-6 所示。图中的一个平面中散落着一些离散点，这些离散点之间有远有近，形成了 4 个类（或称簇）。在同一个类中的节点间距离很近，而与不同类中的点之间距离相对较远。K-means 算法的目标就是在没有明确的分类规则的情况下，将这些点自动分为 4 类。

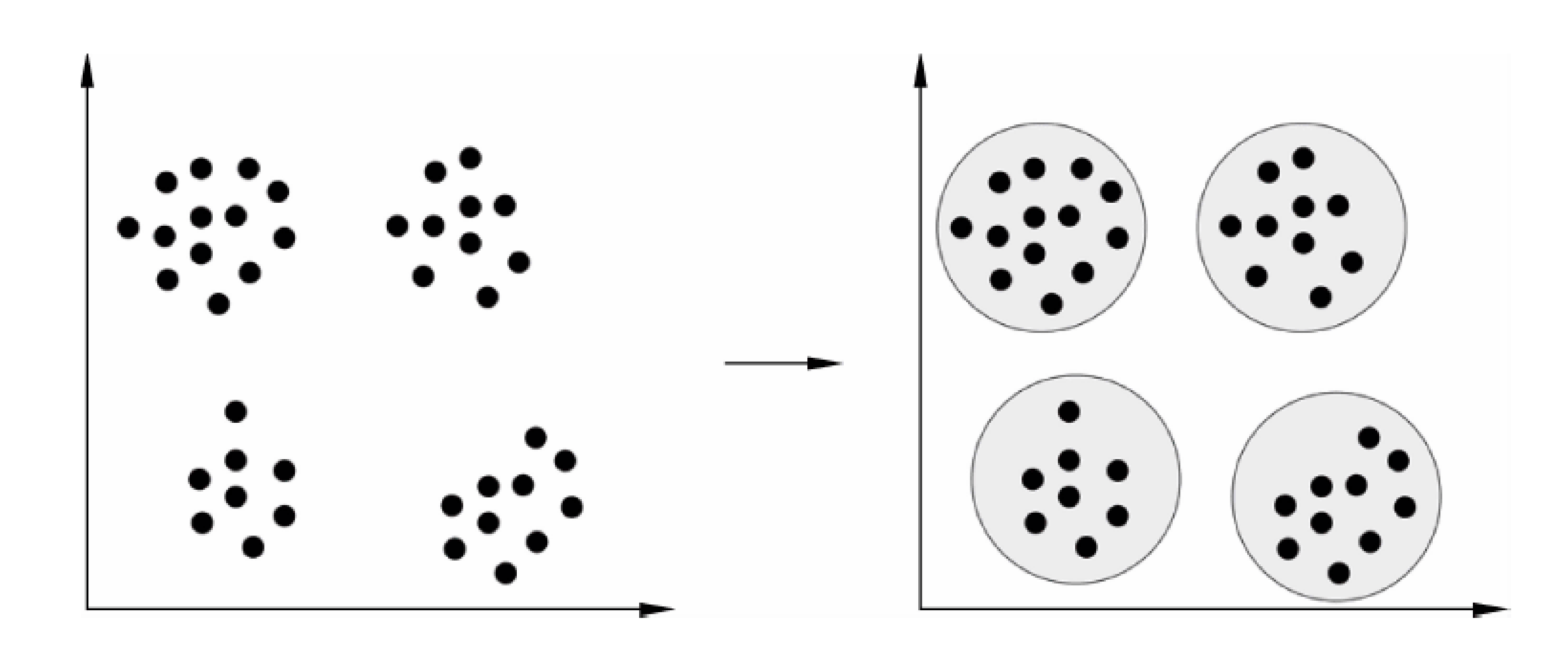


图 1-6 K-means 算法示意图

K-means 算法的基本步骤为：

（1）初始化 K 个点为类的中心；

（2）计算每个样本点到 K 个类中心的距离；

（3）将每个数据点归类到离它最近的那个类中心所属的类；

（4）计算并更新每个类的中心；

（5）重复步骤（2）~（4），直到每个类的中心不再发生变化（或变化值小于一个阈值）。

图 1-7 以将 5 个样本点分为 2 类为例，展示了 K-means 算法的每一步过程，其中 A ~ E 为 5 个样本点，实心圆点为 2 个类的中心点。

从 K-means 的算法过程中可以看出这是一个典型的迭代算法，且在样本点分类的过程中样本点之间是独立的，因此可以很容易地使用 MapReduce 计算模式实现。K-means 的 MapReduce 算法基本思路是实现一个 Job，其中用 Map 函数完成算法中的步骤（2）“计算每个样本点到 K 个类中心的距离”和步骤（3）“将每个数据点归类到离它最近的那个类中心所属的类”。而 Reduce 函数则负责第（4）步“计算并更新每个类

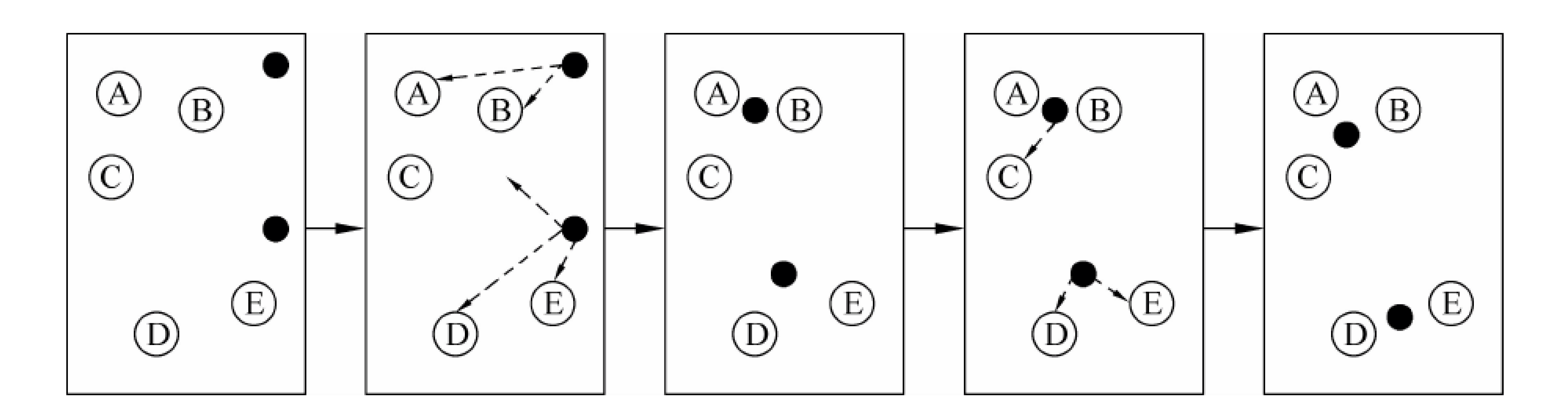


图 1-7 K-means 算法过程图

的中心”。

K－means 算法 Map 函数

```
输入：
    1. centers－K 个类的中心点
    2. dataset－分配由该 Map 函数处理的样本点
输出：
    <key,value>－key 为某样本点所属的类编号,value 为样本点的值
```

```
1:  FOR EACH sample IN dataset DO
2:    class = NULL
3:    min_distance = MAX
4:    FOR EACH center IN centers DO
5:      distance = calculateDistance(sample, center)
6:      IF class == NULL OR distance < min_distance THEN
7:          class = center.class
8:          min_distance = distance
9:      END IF
10:    END FOR
11:  EMIT(class, sample)
12:  END FOR
```

Map 函数的输入有两个，centers 和 dataset。其中 centers 为 K 个类的中心点，在 Job 第一次运行时，这 K 个中心点可由 Main 函数随机初始化，在后续的每轮迭代 Job 运行时，需要从上一轮迭代产生的输出文件中读取出上一轮迭代产生的 K 个类的中心点。这些中心点可以通过 MapReduce 的分布式缓存对象分发到每个 Map 函数中。dataset 为全部样本点数据的一个分片，被分配到一个 Map 函数中进行运算。代码第 1 ~ 10 行，就是寻找 dataset 中每个样本点最近(例如欧式距离最小)的中心点，并将这个样本点归为最近中心点的所属类。代码第 11 行将每个样本点的类编号和样本点的值传递到 Reduce 阶段。

```
K-means 算法 Reduce 函数

输入:
    <key, S> - key 为类编号,S 为属于该类的样本点集合
输出:
    file_center - 存储中心点的文件
    file_dataset - 存储样本点分类结果的文件

center = NULL
FOR EACH sample IN S DO
  center = updateCenter(sample, center)
  APPEND(file_dataset, <key, sample>)
END FOR
APPEND(file_center, center)
```

Map 函数的输出经过 Shuffle 后,属于同一类的样本点会进入同一个 Reduce 函数。Reduce 函数利用同一类的所有样本点完成中心点的更新,并将分类后的样本点和更新后的中心点保存到 HDFS 文件中。在 K-means 算法的 main 函数中,只要对每轮迭代产生的 K 个中心点与上一轮产生的中心点进行比较,如果有一类的两轮中心点距离大于指定阈值,则继续进行下一轮迭代;如果都小于指定阈值,则迭代结束,此时,Reduce 函数输出的聚类结果即为最终结果。当然,为了避免迭代无法停止,我们也可以设定一个最大迭代次数以强制迭代停止。

由上面我们介绍的 K-means 算法的 MapReduce 实现可以看出,每一次迭代过程都需要启动一个完整的 MapReduce 作业,这一启动过程就需要消耗一定的资源。而每一轮作业结束,计算结果都需要写入 HDFS,下一次迭代需要再次从 HDFS 读取该文件,这种模式要消耗大量的磁盘和网络 IO。而以上两个问题是由 Hadoop 本身系统架构决定的,通过任何的算法优化都不能避免。在参考文献[8]中,作者比较了基于 MapReduce 和 Spark 的 K-means 算法程序的性能,如图 1-8 所示。左图显示的是在同等条件下第一次迭代和第一次之后的每次迭代所消耗的时间。由于 MapReduce 每次迭代都需要从磁盘读取和向磁盘写入数据,因此都需要消耗较长的时间,且相差不大。

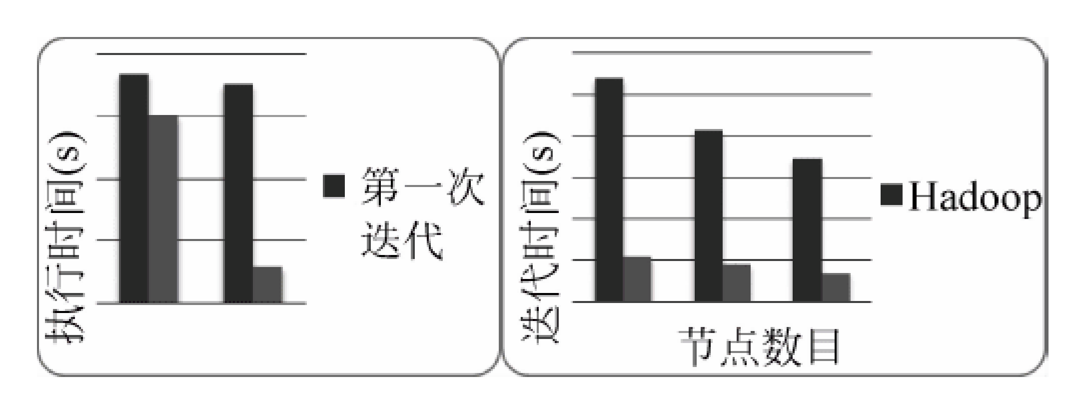


图 1-8　MapReduce 与 Spark 实现 K-means 算法性能的比较

而 Spark 充分发挥了内存的作用,在第一次迭代加载完数据到内存后,后续的每次迭代只需要很少的时间。因此,在右图中可以看到每轮平均迭代时间,Spark 比 MapReduce 有了极大降低。

1.2.3 针对 Hadoop 的改进

针对 Hadoop 自身性能上的不足,Hadoop 领域相关的开发者也尝试在 Hadoop 的基础上进行多方面的改进,试图解决 Hadoop 的性能问题。下面我们以两个比较知名的技术简要介绍下这方面的改进成果及其局限性。

- **Twister:迭代式 MapReduce 计算框架**

Twister 是由美国印第安纳大学开发的一个轻量级迭代式 MapReduce 计算框架。该架构主要由3部分组成:Twister Driver、Broker Network、Twister Daemon。其中 Twister Driver 负责驱动整个系统的 MapReduce 计算。Broker Network 是一个独立的模块,负责所有消息和数据的传递。Twister Daemon 运行在每个工作节点上,负责管理工作节点上的 Map/ Reduce 任务。同时,Twister Daemon 与 Broker Network 相连,接受 Broker Network 的数据和指令。相对于原生的 Hadoop 架构,Twister 在处理迭代运算上主要做了两方面的改进。

(1) 增加 Broker Network 模块提高迭代计算时的数据传输效率。Twister 将 Map 处理的中间结果存放于分布式内存中,并由 Broker Network 传递给 Reduce 任务节点。Reduce 任务完成后,所有由 Reduce task 产生的结果通过一个 combine 操作进行统一归并,归并的结果由代码经过条件判断来决定是否需要进行下次迭代。需要迭代的数据同样通过 Broker Network 传给 Map 任务节点反复进行。

(2) 增加 Task Pool 提高每轮迭代的作业启动效率。为了避免每次迭代都进行耗时较长的创建 Task 操作,Twister 引入了一个 Task Pool。在 Task Pool 中维护了若干已创建且不终止的 Task 实例,每次迭代过程都是从 Task Pool 中取出一个 Task 进行,从而避免创建 Task 时 JVM 启动的性能消耗。

虽然 Twister 在并行迭代算法上进行了大胆的尝试,但 Twister 本质上还是基于 MapReduce 模型的,所以,Twister 不能完全克服 MapReduce 的缺陷。首先,Twister 没有底层的分布式文件系统,中间结果全部存放于分布式内存中。Twister 假设内存足够大,中间数据可以全部放在内存中,但这在实际应用环境中并不现实。其次,Twister 同样只有两个函数(Map/ Reduce),计算模型抽象程度不够。无法适应复杂的计算表述。

- **HaLoop:基于 Hadoop 改进的迭代计算框架**

HaLoop 也是在 Hadoop 基础上扩展而来的,可以说是 Hadoop 的一个变体。为了适

应迭代算法的应用场景,HaLoop 在基本 Hadoop 架构上作了很多改进,包括:

(1) 在主节点增加一个新的迭代控制模块,负责不断启动 MapReduce 计算作业,及根据终止条件判断是否结束迭代。

(2) 专门为迭代运算设计并实现了一个新的 Task Scheduler,并充分利用 Data Locality 特性提高计算速度。

(3) 数据在各个 Task Tracker 中会进行缓存和建立数据索引,提高迭代计算效率。

(4) 改进计算模式,将所有迭代式任务抽象为:$R_{i+1} = R_0 \cup (R_i \bowtie L)$,其中 R_0 为输入,L 为迭代中的不变数据,R_i 为第 i 次迭代结果,同时,为这种抽象模式增加了多个编程接口以简化迭代式算法的开发实现。

虽然 HaLoop 在 Hadoop 的架构上和计算模式上都作了相应的改进,但是依然不能根本解决 Hadoop 的性能问题。首先,在计算模式上,HaLoop 仍然是 Map + Reduce 的循环模式,计算模型抽象程度同样不够高。另外,没有对迭代中重复使用的数据有一个统一的抽象表达。虽然可以将中间重复使用的数据放入缓存,但这些数据对用户仍然是不可见的,用户不能对这些中间结果数据进行自定义的操作,因此,使用场景非常有限。

虽然这些基于 Hadoop 基础的改进最终都失败了,但这些改进中的一些关键技术为后来者提供了很有价值的借鉴,例如中间结果的分布式内存存储方式、新的抽象计算模式等。这些思想和关键技术的积累,最终带来了革命性的 Spark 技术的出现。

1.3 大数据技术新星——Spark

1.3.1 Spark 的出现与发展

2009 年,美国加州大学伯克利分校(University of California Berkeley)AMPLab 实验室的博士研究生 Lester Mackey 在使用 Hadoop 实现机器学习算法设计时也遇到了 Hadoop 编程模型过于简单和执行模式低效的问题。Lester 在使用 Hadoop 设计机器学习算法时发现,他首先要想的不是如何提高算法本身的效率,而是迎合过于简单的 MapReduce 计算模式,并且还无法绕开 MapReduce 的性能缺陷。为此,Lester 向同在 AMPLab 实验室的研究生 Matei Zaharia 寻求帮助。Matei 也是 Hadoop 的一个重要贡献者,对 Hadoop 的运行机制有着深刻的理解。在听取了 Lester 的意见后,Matei 总结了 Hadoop 的不足,并开始设计 Spark[9] 的第一个版本。

Matei 在设计 Spark 时,首要目标就是要避免运算时出现过多的网络和磁盘 IO 开销。因此,他将 Spark 的核心数据结构设计为弹性分布式数据集(Resilient Distributed

Dataset，RDD）。RDD 是 Spark 的核心数据结构，用户可以使用 RDD 将一部分数据集缓存在内存中，使其能在并行操作中被有效的重复使用。Spark 为 RDD 提供了一系列丰富的算子，也就是对 RDD 进行操作的函数。同时，为了避免 Hadoop 架构中启动和调度作业消耗过大的问题，Spark 采用基于有向无环图（Directed acyclic graph，DAG）的任务调度机制进行优化，这样可以将多个阶段的任务串联或者并联执行，无须将每一阶段的中间结果数据存到 HDFS 上。这些原理和机制的具体内容，我们将在后面的章节中详细介绍。

Spark 是踩在 Hadoop 这种已经非常优秀的技术的肩膀上而出现在世人眼中的，它解决了 Hadoop 在复杂运算场景中的性能问题，并以此为基础扩展出了 MLlib 机器学习库、Spark Streaming 流式计算模式和 GraphX 图计算模式，俨然要实现大数据处理多种计算模式的一站式解决方案，因此，很快得到了业界的认可。以前支持 Hadoop 技术的一些大型和新兴公司，例如 Yahoo、Intel、Cloudera、IBM 等，都开始全力使用和推进 Spark 技术。我们梳理了 Spark 技术在最近 5 年发展与演进中的重要事件，以便于大家了解 Spark 技术从简单的技术雏形到完整的技术架构的发展历程。

√ 2009 年：Spark 项目诞生。
√ 2010 年：Spark 对外开源，成为 Apache 社区项目。
√ 2014 年 2 月：Databricks 发布了 Spark 的第一个版本 0.9.0，Spark Streaming 结束了 alpha 版本，同时新增加了 alpha 版本组件 GraphX，MLlib 在这个版本中增加了常用算法，Spark 也成为 Apache 顶级项目。
√ 2014 年 2 月：大数据公司 Cloudera 宣布将以 Spark 框架取代 MapReduce。
√ 2014 年 4 月：Apache Mahout 不再使用 MapReduce 算法支持，全部改用 Spark 作为计算引擎。
√ 2014 年 5 月：Spark 的第一个正式版本 1.0 发布，并推出了 Spark SQL 这个新项目。
√ 2014 年 9 月：Spark 的新版本 1.1.0 发布。
√ 2014 年 12 月：Spark 发布 1.2 版本，GraphX 结束 alpha 版本，对外正式发布。
√ 2015 年 6 月：IBM 宣布投入 3500 名研究人员加入 Spark 社区，并建立 Spark 技术中心。
√ 2016 年 1 月：Spark 发布 1.6 版本，提升性能，并在 DataFrame 之后增加 Dataset API。

我们可以看到，Spark 的发展速度和势头远远超过了已略显疲态的 Hadoop。从目

前的形势看，可能在很快的一段时间内，Spark 将全面取代 Hadoop（尤其是其中的 MapReduce 计算模式）。为了直观地说明这一趋势，我们用两组对比数据来展示。

图 1-9 是 Google 趋势中计算机科学类别下 Spark（点线）和 Hadoop（实线）两个关键词的搜索热度趋势。我们可以看到，虽然从 2009 年至今，Hadoop 的搜索热度整体上还是高于 Spark，但是 Spark 的增长趋势确实明显快于 Hadoop。尤其是在 2015 年 6 月之后，Spark 的关注度已经超过了 Hadoop。

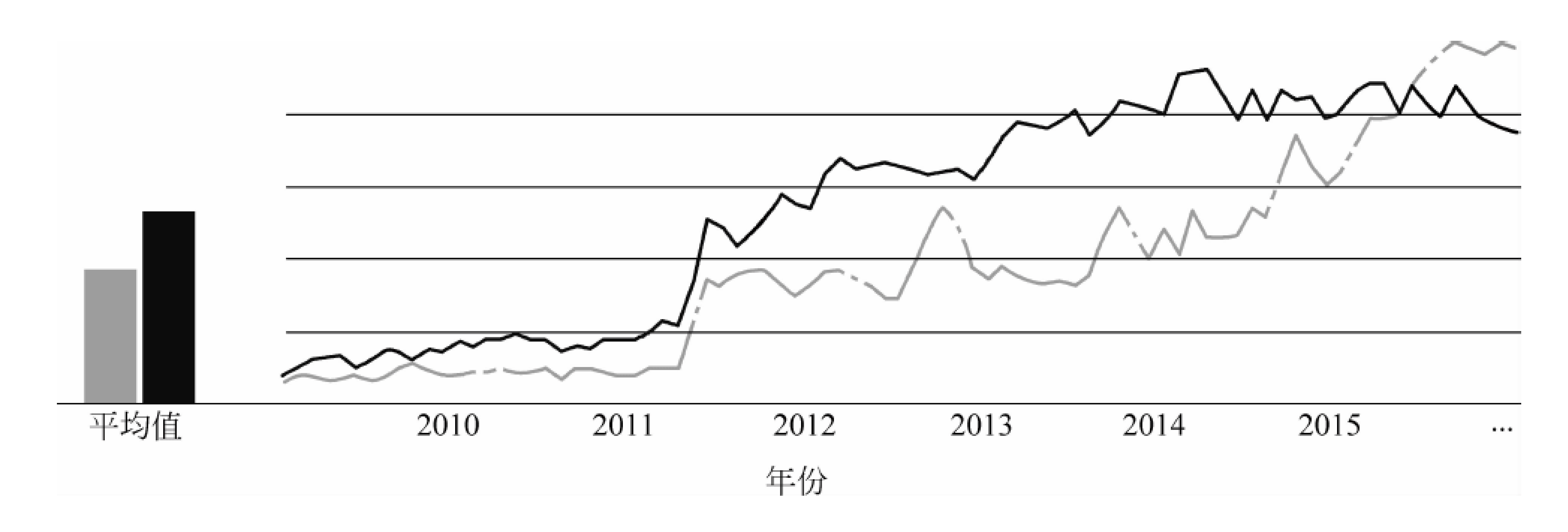


图 1-9　Hadoop 与 Spark 搜索热度图

图 1-10 是 Redmonk 统计的在全球最为活跃的程序员交流社区 Stack Overflow 上大数据相关技术的标签数量，该数量代表了某项技术在程序员中的热度。我们可以看到，Spark 的关注度增长迅猛，并且在 2015 年伊始就显著地高于 MapReduce 了。

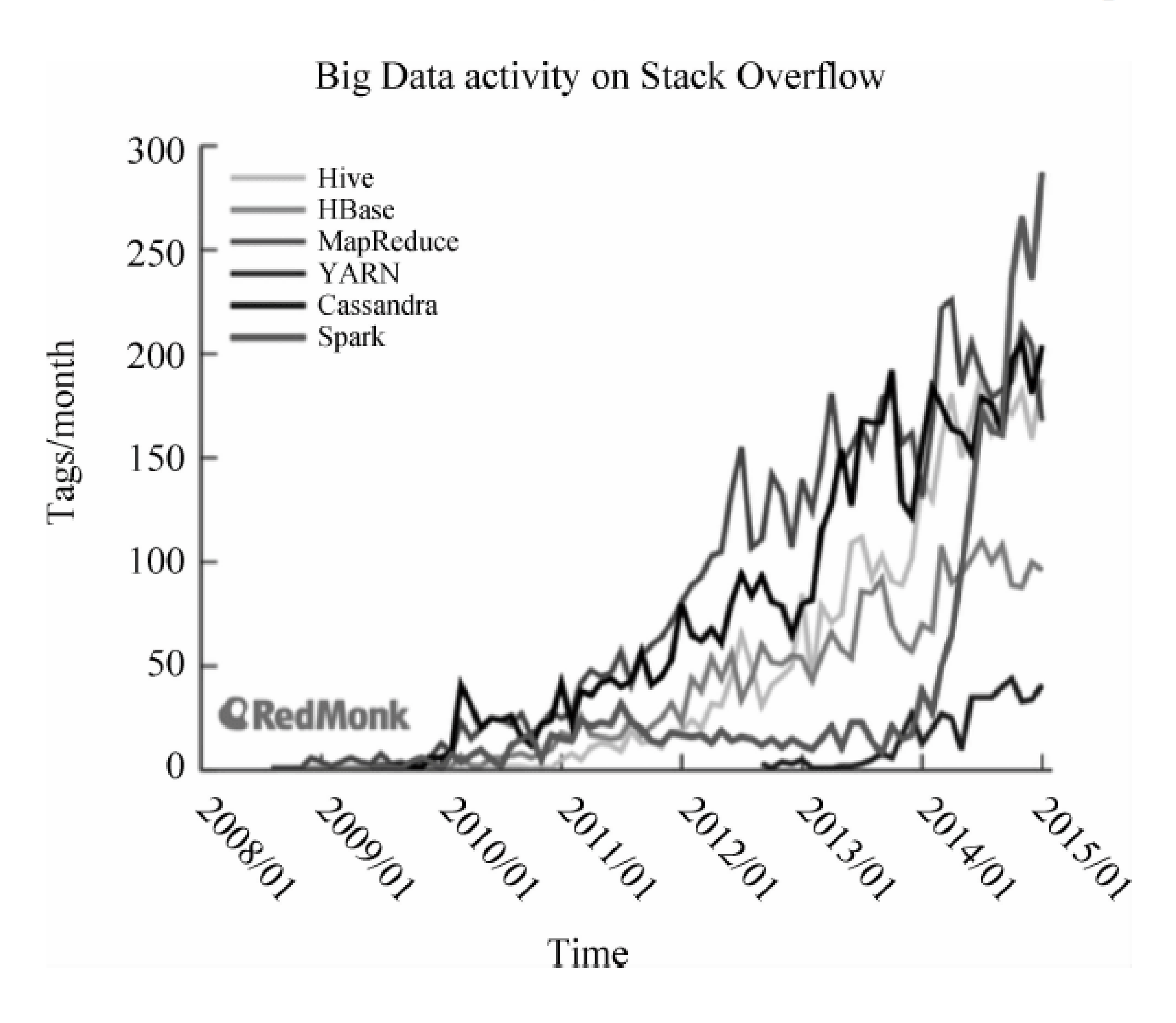


图 1-10　MapReduce 与 Spark 在 Stack Overflow 的热度图

以上这两项数据充分说明了 Spark 已经成为当今大数据领域最为活跃、最为热门的关键性基础技术。

1.3.2 Spark 协议族

AMPLab 实验室在 Spark 项目的初始阶段就将其发展目标定为“One stack to rule them all”，也就是说 Spark 的目标是将批处理、交互式处理、流式处理、各种机器学习算法、图形计算、SQL 查询等全部融合到一个软件栈中，同时，还要与目前成熟的 Hadoop 技术组件有很好的结合。因此，Spark 在其发展过程中不断丰富和完善了一系列支持各类大数据处理场景的技术族，以满足不同业务需求，这些技术族有机构成了一个协议栈，如图 1-11 所示。其中蓝色部分为 Spark 自身技术族，灰色部分为可协同工作的外部技术，受篇幅所限，在这里我们主要介绍 Spark 自身技术，对其他相关技术感兴趣的读者可以查阅相关资料。

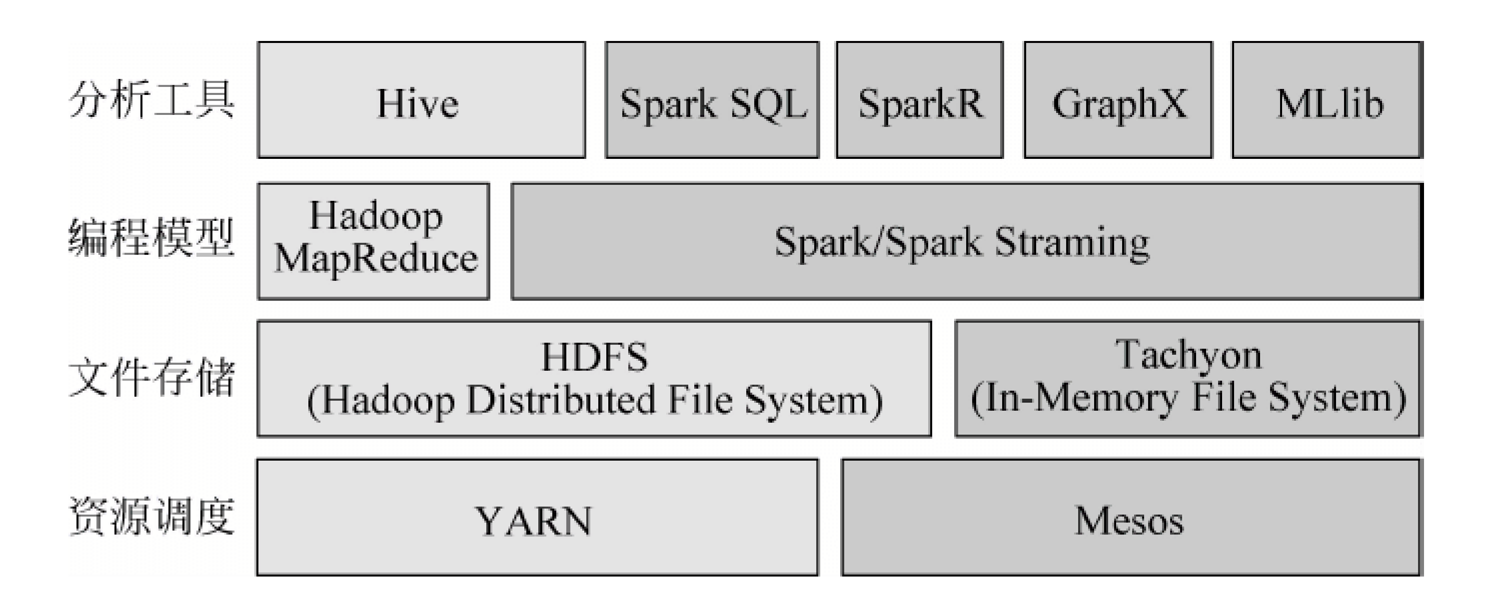


图 1-11 Spark 技术族

- Mesos：Mesos 位于整个协议栈的最底层，负责集群资源的管理。Spark 集群支持运行多种不同的计算框架，例如 Spark、Hadoop、Storm、MPI 等。这就需要对集群资源进行分配、调度和监控，以合理地将资源分配给其上运行的每个框架。Mesos 就是为支持这一目标实现的技术，其提供了跨应用、跨框架的资源隔离和共享，并支持高效的资源任务调度。
- Tachyon：Spark 在设计之初，就支持处理存放在 HDFS 中的数据。同时，Spark 开发团队也考虑到基于磁盘 IO 的 HDFS 文件系统可能很难应对性能要求更高的计算场景，因此，设计和实现了基于分布式内存的高性能文件系统 Tachyon。简单来看，可以将 Tachyon 视为用分布式内存替代磁盘的“新型 HDFS”系统（目前 Tachyon 已更名为 Alluxio）。
- Spark Core：Spark Core 是 Spark 整个计算平台的计算执行引擎，实现了包括任务调度、内存管理、故障修复、序列化和压缩在内的核心基础功能。Spark Core 将分布式数据统一抽象为弹性分布式数据集（Resilient Distributed Datasets，RDD），并实现了多种构建和操作 RDD 的 API 接口。Spark Core 采用函数式编

程语言 Scala 编写而成，同时支持 Scala、Python、Java 编写 Spark 程序。

- Spark Streaming：Spark Streaming 是 Spark 处理大规模流式数据的技术组件。Spark Streaming 按时间片将不断到达的流式数据构建成小片的 RDD，然后以细时间粒度的批处理模式实现流式计算。
- Spark SQL：Spark SQL 是 Spark 处理结构化数据的技术组件。通过 Spark SQL，可以使用 SQL、HQL(Hive Query Language)中的关系型查询表达式对数据进行处理。Spark SQL 支持多种类型的数据格式，包括 Hive 表、JSON、Parquet 格式文件等。
- MLlib：MLlib 是基于 Spark 的一个并行机器学习算法库。MLlib 提供了多种类型的机器学习算法，包括分类、聚类、回归、协同过滤等。
- GraphX：GraphX 是一个支持大规模图计算的并行计算库。图的表达和运算有其特殊性，因此，GraphX 在原生 Spark RDD 的 API 上进行了扩展，提供更加适合图计算的 API，并实现了多种图算法，包括 PageRank、Triangle 计算等。
- SparkR：R 语言是目前最为流行的数据分析开源软件之一，但 R 语言的单机运行环境能够计算的数据量有限。为了支持掌握 R 语言的大量分析人员利用 Spark 的能力处理海量数据，Spark 技术团队提供了 SparkR 技术组件，用户可以通过 R 中的指令和函数对 Spark RDD 进行操作，SparkR 负责将这些 R 语句翻译成可执行的 Spark 程序，完成对海量数据的处理。

1.3.3 Spark 的应用及优势

Spark 推出后，由于其高性能和高灵活性的优点，被很多公司和机构用于替代 MapReduce 进行数据挖掘和机器学习等相关工作。在这里，我们以 Twitter 应用 Spark 进行数据分析的实例来简要展示一下 Spark 的优势。

Twitter 是一个提供社交网络服务的网站，2014 年的 Twitter 公布的公开数据显示，该网站拥有超过 2.4 亿的月活跃用户数，这些用户每天会发表约 5 亿条推文，每天需处理约 16 亿的网络搜索请求，这些访问和操作每天会产生超过 100TB 的压缩数据。Twitter 的数据分析师每天要对这些海量数据进行深入分析，以支持广告业务优化、搜索优化、好友和内容推荐，以促进公司的用户和收入增长。具体来说，在 Twitter 上，当一个用户关注另一个用户时会产生一条“Follow”记录。对于 Twitter 来说，用户之间的关注关系对于用户的分类、协同过滤等都具有重要的意义，因此，利用存储的数据对关注关系进行运算是 Twitter 中的重要基础计算。下面我们以一个最基础的计算实例来

说明 Twitter 使用 Spark 后获得的好处。

Twitter 中的一个功能是提供被关注用户数的用户排行榜,即按关注某用户的用户数从大到小排序。为了计算这个排行,需要用到两个数据源:①“Raw Users”,记录了用户标识、使用的语言等个人信息;②“Raw Follows”,记录了用户间的关注关系,包括关注者的标识和被关注者的标识。在 Spark 出现之前,Twitter 公司是使用 Hadoop 技术族中的 Pig 脚本为分析工具,计算过程如图 1-12 所示。

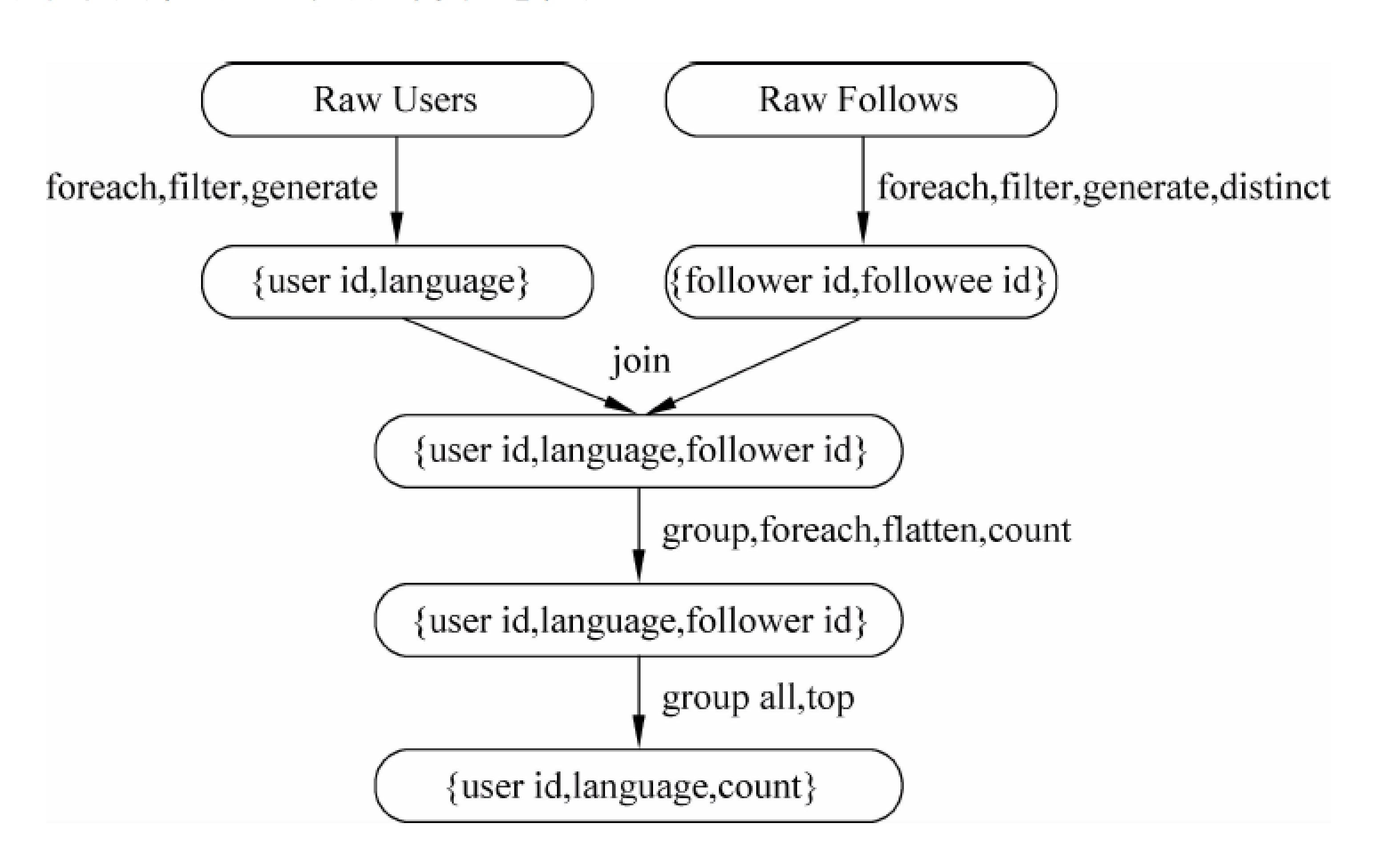


图 1-12 Twitter 使用 Pig 处理社交关系图

Pig 脚本首先要将“Raw Users”和“Raw Follows”按照需求条件进行过滤(filter)、去重(distinct)等操作,然后对两类数据进行联结(join)操作,然后对联结后的数据进行统计得到排行榜。这其中的每一步 Pig 操作都将转化为多个 MapReduce 作业,这些 MapReduce 作业都要从 HDFS 上读取数据再将结果输出到 HDFS,然后进行下一个作业。虽然 Pig 在将操作转化为 MapReduce 作业上做了多方面的优化,但 Twitter 在处理 PB 级原始数据时仍然需要花费大量的时间。在 Spark 出现后,Twitter 公司转而使用 Spark 对用户关注关系进行分析,对于上面同样的分析目标,Spark 的计算过程如图 1-13 所示。

从计算过程我们可以看到,由于 Spark 提供丰富的计算算子,使得 Spark 对数据的操作更加简洁明了。另外,在进行 filter、join、reduceByKey 等操作时,Spark 采用“惰性”求值的方式,即对数据的操作并不立即执行,而是仅仅记录下转换操作的数据对象。只有当需要有返回结果时,才真正执行。这就使得 Spark 在进行数据操作时可以大大减少数据的读取操作,从而提高运行效率。图 1-14 展示了 Twitter 测试获得的 Spark 和 Pig 的计算性能对比。Twitter 使用了一小一大两个数据集,其中 Raw Users 数据均为 430GB,而 Raw Follows 数据则分别为 15GB 和 466GB。对于每次测试,记录了任务完成

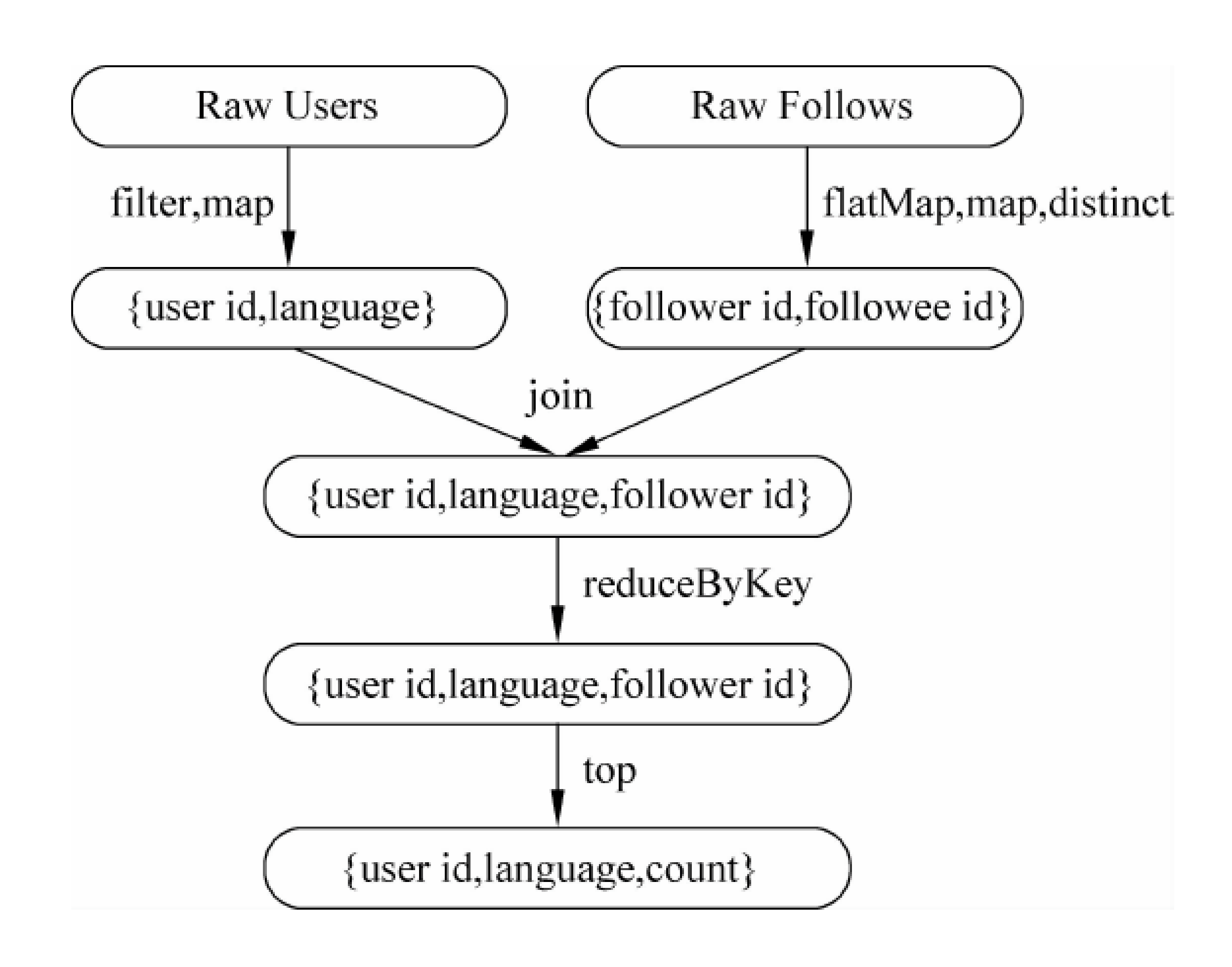


图 1-13 Twitter 使用 Spark 处理社交关系图

所消耗的 CPU 时间，及用户实际感知的 Wall-Clock 时间。从结果图中我们可以看到，相比性能已经在原始 MapReduce 程序基础上进行了大量优化的 Pig，Spark 仍然能带来 2 倍以上的性能提升。

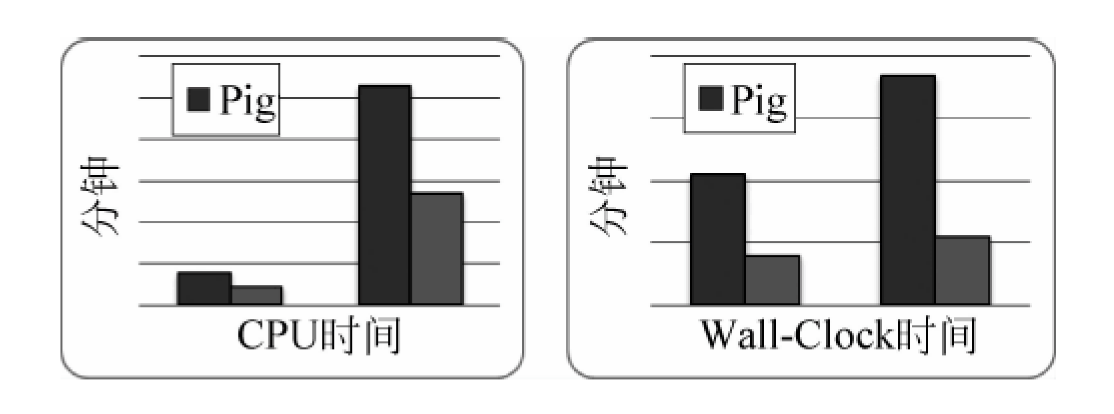


图 1-14 Twitter 使用 Spark 后的性能提升

第2章

体验 Spark

在上一章我们简要介绍了将人们引入大数据时代的火种——Hadoop 技术，以及这项技术在处理复杂运算时的局限性。同时，我们通过一个简单的 K-means 算法对比，展示了 Spark 相比 Hadoop 的优势。相信大多数人看到这里，都会“食指大动”想尝一尝 Spark 的鲜了。与 Hadoop 复杂的安装过程不同，Spark 提供了一个非常简单的单机运行环境供初学者使用。在这一章里，我们就带着大家一步一步地从安装和使用 Spark 到编写和运行 Spark 程序，再到查看 Spark 的运行状况，亲身体验下简洁高效的 Spark 技术。

2.1 安装和使用 Spark

在本节中，我们主要介绍如何搭建单机版的 Spark，用于学习和调试 Spark 程序。因为我们并不会用这个学习环境进行大数据量的处理，因此，可以不需要安装 Cygwin、HDFS 等环境。相比安装复杂的 Hadoop，Spark 单机版环境的安装非常简单。可以毫不夸张地说，在安装文件已下载就绪的前提下，你可以在 5 分钟内快速搭建一个单机版的 Spark 环境进行体验。

2.1.1 安装 Spark

Spark 的设计目标是部署于运行 Linux 的服务器集群进行大数据处理，因此，在 Linux 环境下的安装非常简单。而 Mac OS X 系统是基于 Unix 开发的，安装 Spark 的过程也很简单。相比之下，要在目前使用人数最多的 Windows 系统上运行 Spark 反而略显复杂，所以在本节中我们主要以 Windows 环境为例介绍 Spark 的安装。整个安装过程主要分为4 个步骤：安装 JDK，安装 Scala，安装 Spark，安装 WinUtil。在 Linux 和 Mac OS X 下安装 Spark 只需要完成前 3 步即可。

1. 安装 JDK

Spark 采用 Scala 语言编写,而 Scala 程序是以 JVM 为运行环境的,因此,需先安装 JDK 以支持 Spark 的运行。Spark 通常需要 JDK 6.0 以上版本,你可以在 Oracle 的 JDK 官网上(http:// www. oracle. com/ technetwork/ java/ javase/ downloads/ index. html)下载相应版本的 JDK 安装包,如图 2-1 所示。需要注意的是,在下载时应选择"JDK"安装包,而不是"JRE"。在我们这个示例中,我们选择的是 JDK 7,下载后运行二进制可执行文件,按照默认配置完成安装即可。



图 2-1　JDK 下载页面

2. 安装 Scala

刚才我们提到,Spark 是采用 Scala 语言编写的,因此第二步是要安装 Scala。Scala 官网的下载页面(http:// www. scala-lang. org/ download/)提供了多个版本的 Scala 下载。由于 Scala 各个版本之间兼容性并不好,因此下载时一定要注意你要安装的 Spark 版本所依赖的 Scala 版本,以免遇到一些难以预知的问题。在我们的例子中,是要安装

目前最新的 Spark 1.3.0 版本,因此,我们选择下载所需的 Scala 2.10.4 版本。选择之前的历史版本下载,需要先从下载页面中点击"All previous Scala Releases"链接,进入历史版本列表,然后选择"2.10.4"版本下载(http:// www.scala-lang.org/ files/ archive/ scala-2.10.4.msi)。下载后按照提示一步一步执行安装即可。

√ 在 Windows 中执行命令"cmd",启动 Windows 命令行环境。

√ 在命令行环境中,输入"scala",然后敲回车。

√ 如果看到如图 2-2 所示成功启动 Scala Shell 环境,则说明安装成功,然后输入"exit",退出 Scala Shell 环境。

√ 如果启动 Scala Shell 环境失败,一般只需要在 Windows 环境变量设置界面配置 SCALA_HOME 环境变量为 Scala 的安装路径即可。



图 2-2 Scala 安装成功验证

3. 安装 Spark

Spark 官网(http:// spark.apache.org/ downloads.html)提供了各个版本的安装包。为搭建学习试验环境,我们选择下载预编译好的安装包,例如 spark-1.3.0-bin-hadoop2.4.tgz,如图 2-3 所示。

4. 安装 winutils

由于 Spark 的设计和开发目标是在 Linux 环境下运行,因此,在 Windows 单机环境(没有 Hadoop 集群的支撑)中运行会遇到 winutils 的问题(一个相关的 Issue 可以参见 https:// issues.apache.org/ jira/ browse/ SPARK-2356)。为了解决这一问题,我们需要安装 winutils.exe,具体方法如下。

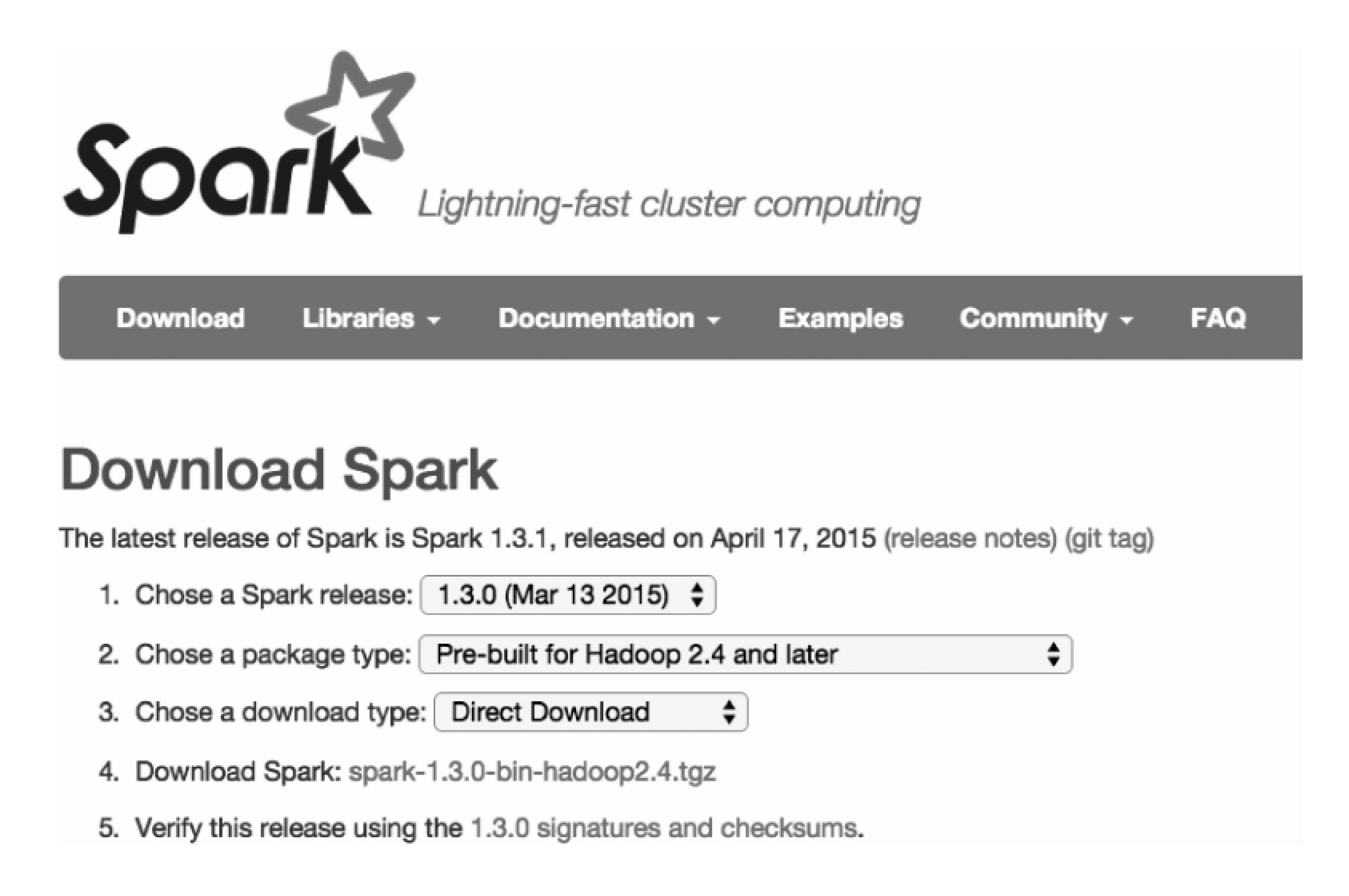


图 2-3　Spark 下载

√ 从一个可靠的网站下载 winutils. exe(我们选择 Hadoop 商业发行版 Hortonworks 提供的下载链接 http:// public-repo-1. hortonworks. com/ hdp-win-alpha/ winutils. exe)。

√ 将 winutil. exe 拷贝到一个目录,例如: E:\LearnSpark\win\bin。

√ 按照如图 2-4 和 2-5 的步骤,设置 Windows 系统的环境变量 HADOOP_HOME 为 E:\LearnSpark\win(注意没有 bin)。



图 2-4　进入 Windows 高级系统设置

至此,在 Windows 下安装 Spark 的过程全部完成。

2.1.2　了解 Spark 目录结构

Spark 安装后,会在安装目录下生成一系列的目录,其中的一些重要目录为:

√ bin 目录下是使用 Spark 时常用的一些执行程序,例如我们进行 Spark 命令交互环境使用的 spark-shell。

√ conf 目录下存放的是运行 Spark 环境所需的配置文件。



图 2-5 配置 Windows 环境变量

√ data 目录下 mllib 需要的一些测试数据。

√ ec2 目录是在 AWS 上部署使用的一些相关文件。

√ examples 目录中有一些例子的源代码和测试文件。

√ lib 目录下存放的是 Spark 使用的一些库，我们之后开发 spark 应用，也是需要使用这些库的。

√ python 目录是使用 python 相关的一些资源。

√ sbin 目录是搭建 Spark 集群所需要使用的一些脚本。

2.1.3 使用 Spark Shell

就像 HelloWorld 程序基本已成为学习某一门开发语言的第一个入门程序一样，WordCount 程序就是大数据处理技术的 HelloWorld。下面我们就以使用 Spark 统计一个文件中的单词出现次数为例，快速体验一下便捷的 Spark 使用方式。

- **启动 Spark Shell 环境**

在 Windows 文件管理器中，切换目录到 Spark 安装后生成的 spark-1.3.0-bin-hadoop2.4 目录下，按住 Shift 键的同时点击鼠标右键，然后使用左键点击“在此处打开命令窗口”。在打开一个命令行的窗口中，输入“bin\spark-shell”，就可以启动 spark-shell 环境，如图 2-6 所示。

如果不希望这么麻烦地切换目录，而是希望在打开一个命令行窗口中直接运行

```
bin\spark-shell
E:\LearnSpark\spark-1.3.0-bin-hadoop2.4>bin\spark-shell
log4j:WARN No appenders could be found for logger (org.apache.hadoop.metrics2.li
b.MutableMetricsFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/05/04 09:47:08 INFO SecurityManager: Changing view acls to: betatheta
15/05/04 09:47:08 INFO SecurityManager: Changing modify acls to: betatheta
15/05/04 09:47:08 INFO SecurityManager: SecurityManager: authentication disabled
; ui acls disabled; users with view permissions: Set(betatheta); users with modi
fy permissions: Set(betatheta)
15/05/04 09:47:08 INFO HttpServer: Starting HTTP Server
15/05/04 09:47:08 INFO Server: jetty-8.y.z-SNAPSHOT
15/05/04 09:47:09 INFO AbstractConnector: Started SocketConnector@0.0.0.0:51851
15/05/04 09:47:09 INFO Utils: Successfully started service 'HTTP class server' o
n port 51851.
Welcome to
                                    version 1.3.0
```

图 2-6 启动 Spark Shell

spark-shell，那么，只需要在 Windows 环境变量中将上面的 spark-shell 所在的路径加入环境变量 PATH 中即可。

- **建立待统计的单词文件**

选择一个已存在的文本文件，或新建一个文本文件，作为待统计的单词文件 E:\LearnSpark\word.txt，在这里我们新建一个文件，内容为：

```
apple banana
banana banana
```

- **加载单词文件**

执行 Spark 程序需要一个 SparkContext 类实例，在 Spark Shell 中已经默认将 SparkContext 类初始化为对象实例 sc。因此，我们不需要再去初始化一个新的 sc，直接输入以下命令使用即可：val file = sc.textFile("E:\\LearnSpark\\word.txt")。

该行命令使用 SparkContext 类的 textFile 函数，加载待统计的单词文件，结果如图 2-7 所示。

- **统计单词出现次数**

如果你用 MapReduce 计算框架编写过 WordCount 程序，那你一定能体会到执行一个简单的单词统计功能需要数十行代码的不便。而利用 Spark 的函数式编程模式，我们只需要一行 Scala 语句即可完成单词统计功能。

```
scala> val file = sc.textFile("E:\\LearnSpark\\word.txt")
15/05/04 09:50:15 INFO MemoryStore: ensureFreeSpace(159118) called with curMem=0
, maxMem=278019440
15/05/04 09:50:15 INFO MemoryStore: Block broadcast_0 stored as values in memory
 (estimated size 155.4 KB, free 265.0 MB)
15/05/04 09:50:15 INFO MemoryStore: ensureFreeSpace(22692) called with curMem=15
9118, maxMem=278019440
15/05/04 09:50:15 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in
memory (estimated size 22.2 KB, free 265.0 MB)
15/05/04 09:50:15 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on l
ocalhost:51876 (size: 22.2 KB, free: 265.1 MB)
15/05/04 09:50:15 INFO BlockManagerMaster: Updated info of block broadcast_0_pie
ce0
15/05/04 09:50:15 INFO SparkContext: Created broadcast 0 from textFile at <conso
le>:21
file: org.apache.spark.rdd.RDD[String] = E:\LearnSpark\word.txt MapPartitionsRDD
[1] at textFile at <console>:21
```

图 2-7 加载单词文件

```
scala > val counts = file.flatMap(line => line.split(" ")).map(word =>
(word, 1)).reduceByKey(_ + _)
```

这行代码的运行结果如图 2-8 所示。在这里我们暂时先不解释这行代码的具体含义，留待在后面的章节中慢慢学习。你只需要体会到 Spark 是如何大幅简化数据处理工作的难度即可。

```
scala> val counts = file.flatMap(line => line.split(" ")).
     |                          map(word => (word, 1)).
     |                          reduceByKey(_ + _)
15/05/04 09:51:58 INFO FileInputFormat: Total input paths to process : 1
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey
at <console>:25
```

图 2-8 统计单词出现次数示例代码

- **保存结果文件**

在这里我们使用“E:\LearnSpark\counts.txt”作为输出文件。需要注意的是，要保证没有和输出文件同名的文件或者是文件夹，如果存在则需要手动删除该文件夹，否则会出错。保存结果文件的命令如下所示：

```
scala > counts.saveAsTextFile("E:\\LearnSpark\\counts.txt")
```

这行代码的运行过程如图 2-9 所示。

下面我们来看一下最后的输出结果，count.txt 其实是个目录，在该目录下有好几个文件，其中 part-00000 和 part-00001 是我们需要的结果。

```
part - 00000 中的内容：(apple,1)
part - 00001 中的内容：(banana,3)
```

```
scala> counts.saveAsTextFile("E:\\LearnSpark\\counts.txt")
15/05/04 09:53:15 INFO deprecation: mapred.tip.id is deprecated. Instead, use ma
preduce.task.id
15/05/04 09:53:15 INFO deprecation: mapred.task.id is deprecated. Instead, use m
apreduce.task.attempt.id
15/05/04 09:53:15 INFO deprecation: mapred.task.is.map is deprecated. Instead, u
se mapreduce.task.ismap
15/05/04 09:53:15 INFO deprecation: mapred.task.partition is deprecated. Instead
, use mapreduce.task.partition
15/05/04 09:53:15 INFO deprecation: mapred.job.id is deprecated. Instead, use ma
preduce.job.id
15/05/04 09:53:15 INFO SparkContext: Starting job: saveAsTextFile at <console>:2
6
15/05/04 09:53:15 INFO DAGScheduler: Registering RDD 3 (map at <console>:24)
15/05/04 09:53:15 INFO DAGScheduler: Got job 0 (saveAsTextFile at <console>:26)
with 2 output partitions (allowLocal=false)
15/05/04 09:53:15 INFO DAGScheduler: Final stage: Stage 1(saveAsTextFile at <con
sole>:26)
15/05/04 09:53:15 INFO DAGScheduler: Parents of final stage: List(Stage 0)
15/05/04 09:53:15 INFO DAGScheduler: Missing parents: List(Stage 0)
15/05/04 09:53:15 INFO DAGScheduler: Submitting Stage 0 (MapPartitionsRDD[3] at
map at <console>:24), which has no missing parents
15/05/04 09:53:15 INFO MemoryStore: ensureFreeSpace(3744) called with curMem=181
810, maxMem=278019440
15/05/04 09:53:15 INFO MemoryStore: Block broadcast_1 stored as values in memory
 (estimated size 3.7 KB, free 265.0 MB)
15/05/04 09:53:15 INFO MemoryStore: ensureFreeSpace(2650) called with curMem=185
554, maxMem=278019440
15/05/04 09:53:15 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in
memory (estimated size 2.6 KB, free 265.0 MB)
15/05/04 09:53:15 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on l
ocalhost:51876 (size: 2.6 KB, free: 265.1 MB)
```

```
15/05/04 09:53:15 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have
all completed, from pool
15/05/04 09:53:15 INFO DAGScheduler: Job 0 finished: saveAsTextFile at <console>
:26, took 0.613925 s
```

图 2-9 输出统计结果过程

2.2 编写和运行 Spark 程序

通过前面体验 Spark Shell,我们已经感受到了 Spark 的魅力所在。但是,使用 Shell 输入 Scala 语句可以解决一些简单的问题,如果要实现复杂的算法,还是需要编写 Spark 程序。下面我们就来介绍下如何使用 Eclipse 来搭建编写 Spark 程序的开发环境以及编译、打包和运行 Spark 程序的方法。

2.2.1 安装 Scala 插件

因为 Eclipse 的安装非常简单,只需要从 Eclipse 官网(http:// eclipse. org/)下载解压即可使用,因此,我们假定在您的电脑上已经安装好 Ecplise 了。然而,考虑到 Scala 插件和 Eclipse 版本之间的兼容性,我们推荐下载 Kelper 版本的 Eclipse,尤其是不要使用 Luna 版本的 Eclipse。虽然 Spark 支持使用 Java、Python 编写程序,但由于 Spark 原生开发语言是 Scala,基于 Scala 的 Spark API 是最完善、更新最快的,因此,我们建议优先选择 Scala 编写 Spark 程序。Eclipse 默认版本是不支持 Scala 的,因此,需

要安装 Eclipse 的 Scala 插件，其步骤如下：

- 在 Eclipse 中选择菜单栏中的“Help”菜单项，然后点击“Install New software”，如图 2-10 所示。
- 在打开的输入框中，输入如下的下载地址，然后输入回车，如图 2-11 所示。

```
http://download.scala-ide.org/sdk/helium/
e38/scala210/stable/site
```

- 选择前两个软件包，然后点击“Next”开始安装 Scala 插件，如图 2-12 所示。

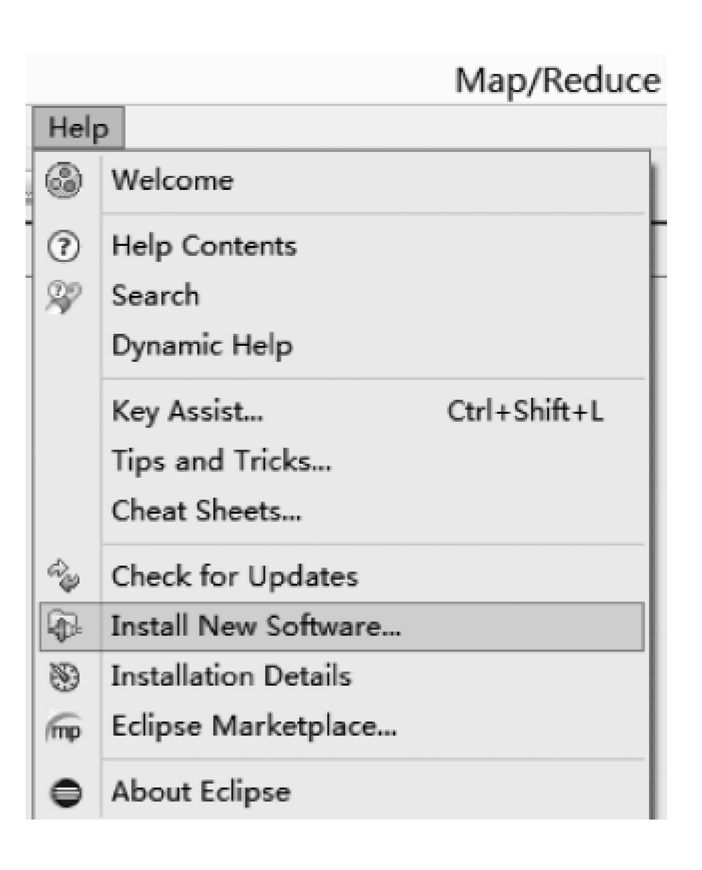


图 2-10　Eclipse 安装新插件

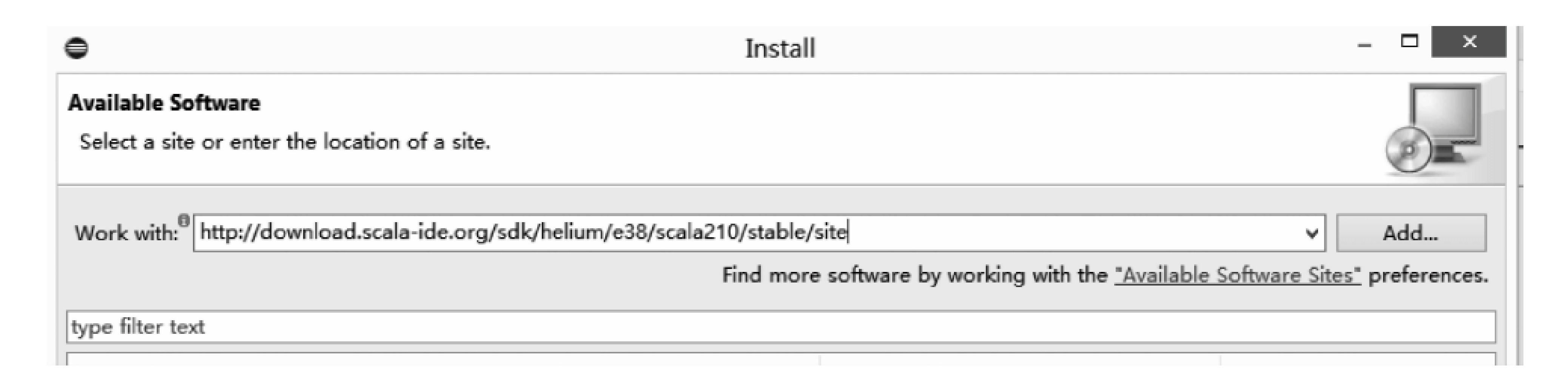


图 2-11　输入下载地址

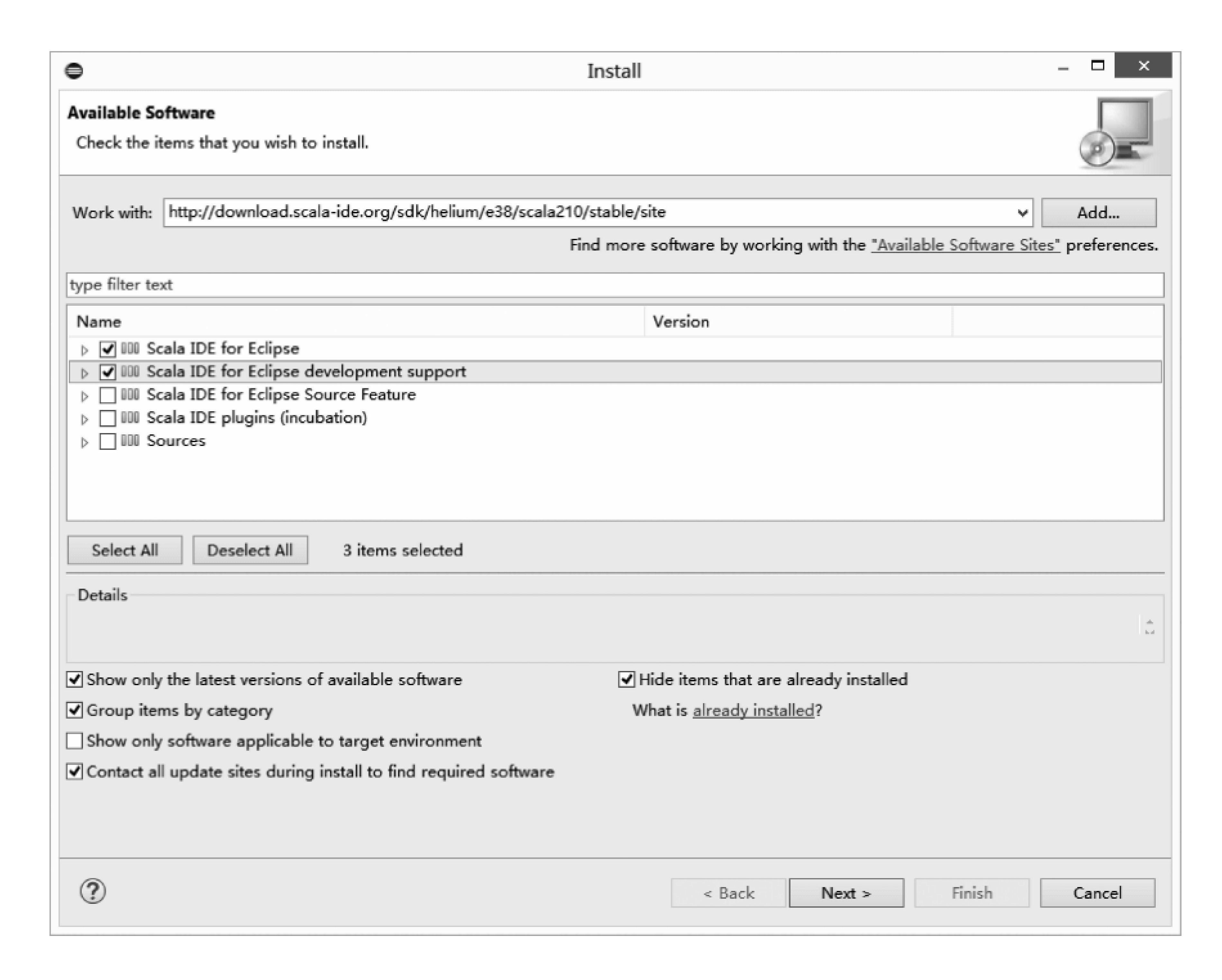


图 2-12　选择软件包

- 安装好之后，重启 Eclipse，然后选择 Eclipse 菜单栏中的“Window”菜单项，选择“Open Perspective”，再选择“Other”，如图 2-13 所示。

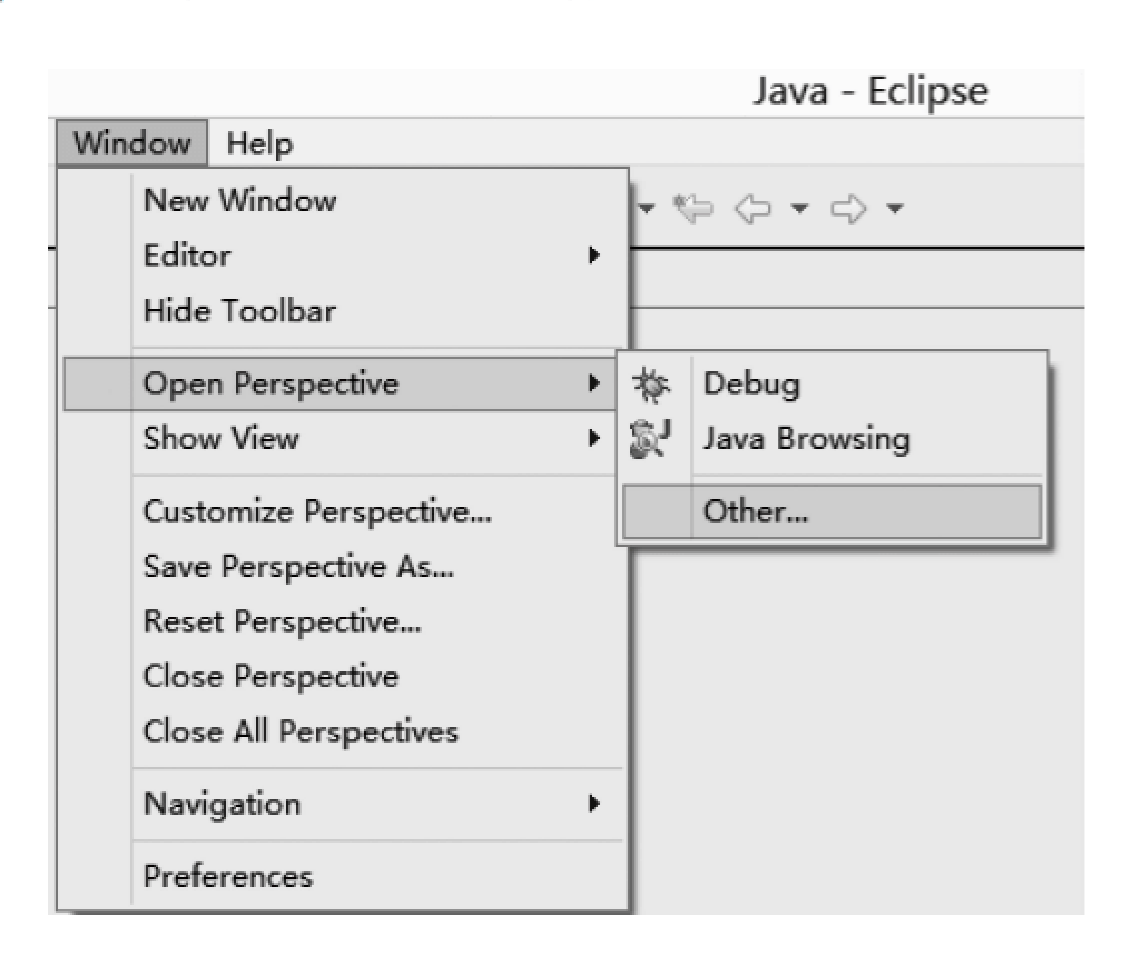


图 2-13 打开新视图

- 在弹出的对话框中选择“Scala”，进入 Scala 视图，如图 2-14 所示。

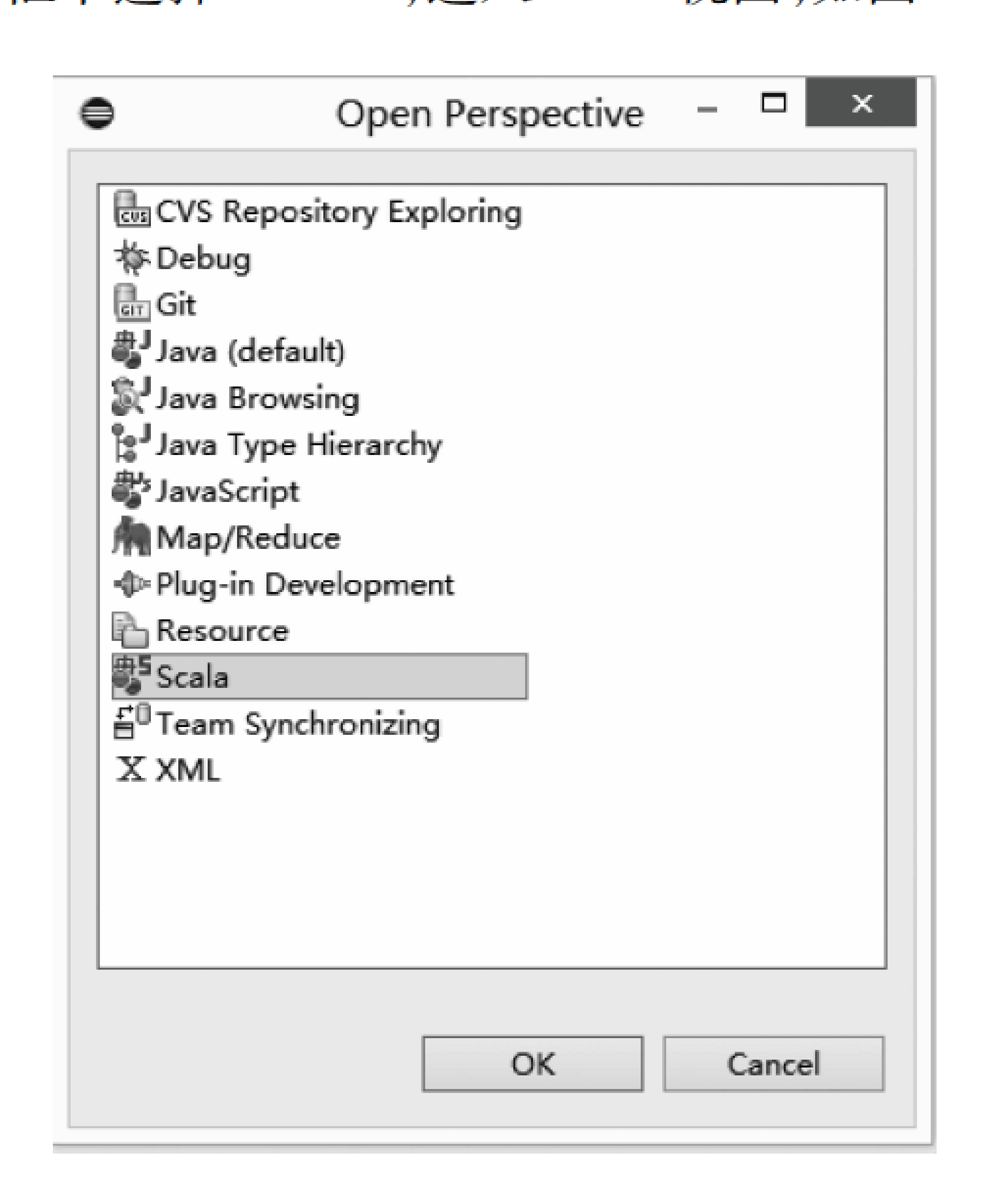


图 2-14 进入 Scala 视图

2.2.2 编写 Spark 程序

编写 Spark 程序，总体可以分为 3 步：首先，新建一个 Scala 工程，然后，添加 Spark 的依赖库，最后，编辑符合需要的源代码。

- 选择菜单栏中的“File”菜单项，从下拉菜单中选择“New”，然后点击“Scala

Project”菜单项,新建 Scala 工程,如图 2-15 所示。

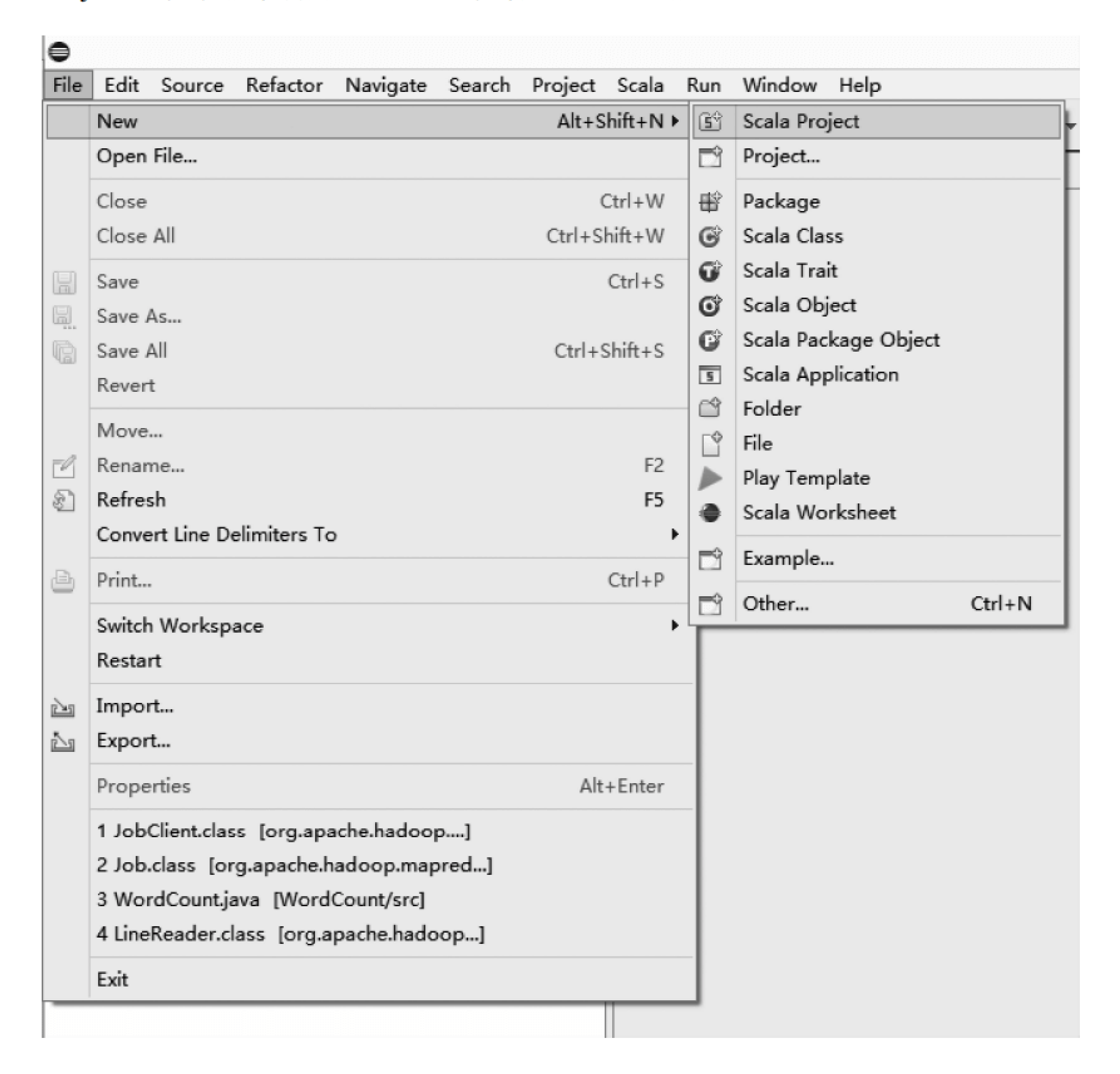


图 2-15　新建 Scala 工程

- 输入新建的 Scala 工程名字,例如“SimpleApp”,然后点击“Finish”按钮,如图 2-16 所示。
- 右键点击新建的 Scala 工程“SimpleApp”,选择“Build Path”菜单项中的“Configure Build Path...”,如图 2-17 所示。
- 在弹出的对话框中选择“Java Build Path”列表项,然后点击“Add External JARs”按钮添加 Spark 依赖库。将目录“spark-1.3.0-bin-hadoop2.4\lib\”下的“spark-assembly-1.3.0-hadoop2.4.0.jar”文件添加到“Java Build Path”中,如图 2-18 所示。
- 添加成功后可以在“SimpleApp”工程列表下看到如下的效果,如图 2-19 所示。
- 右键点击“SimpleApp”工程下的“src”目录,选择“New”菜单项下的“Scala Object”菜单项添加源代码,如图 2-20 所示。



图 2-16　输入 Scala 工程名称

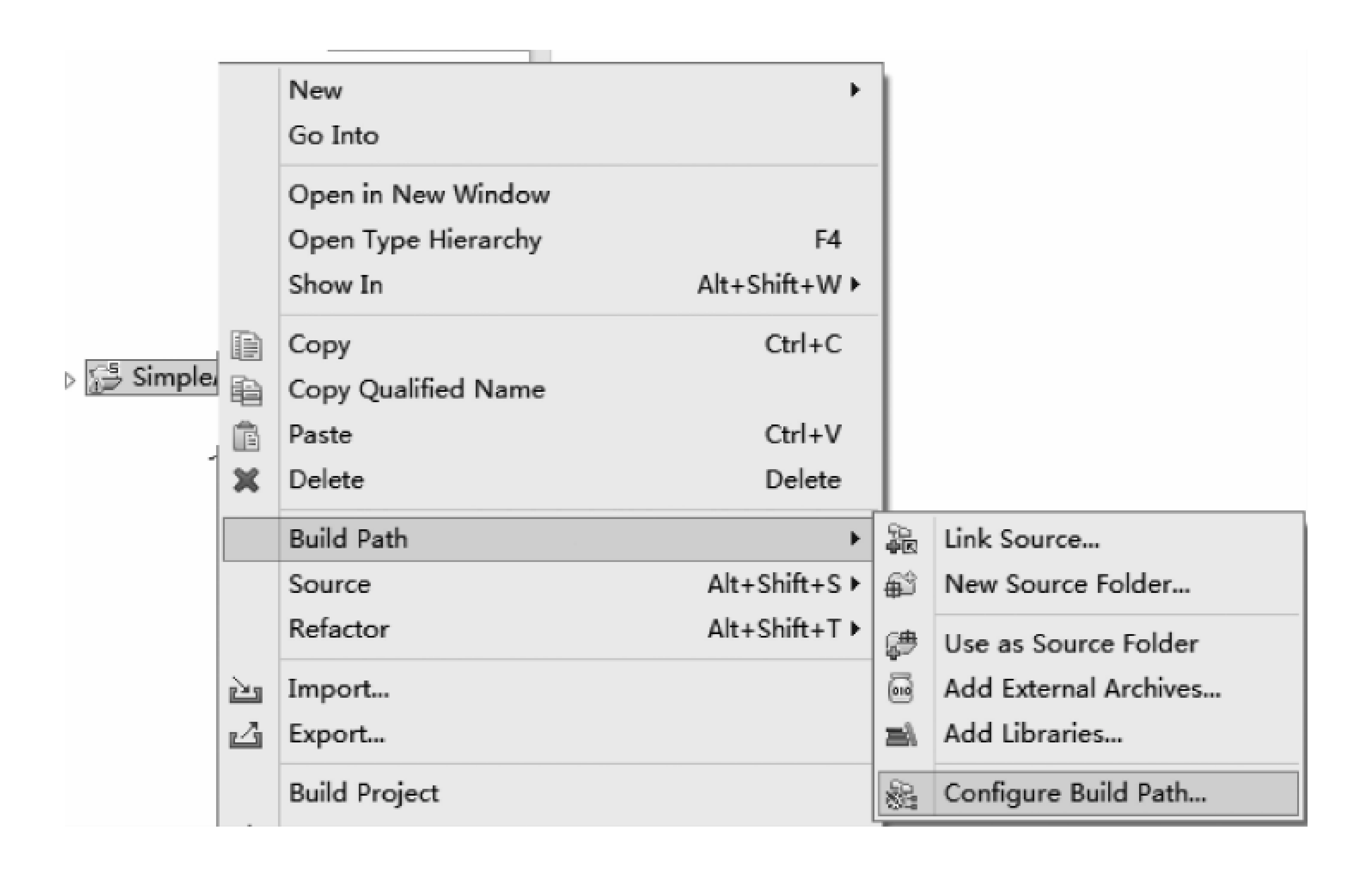


图 2-17　配置编译地址

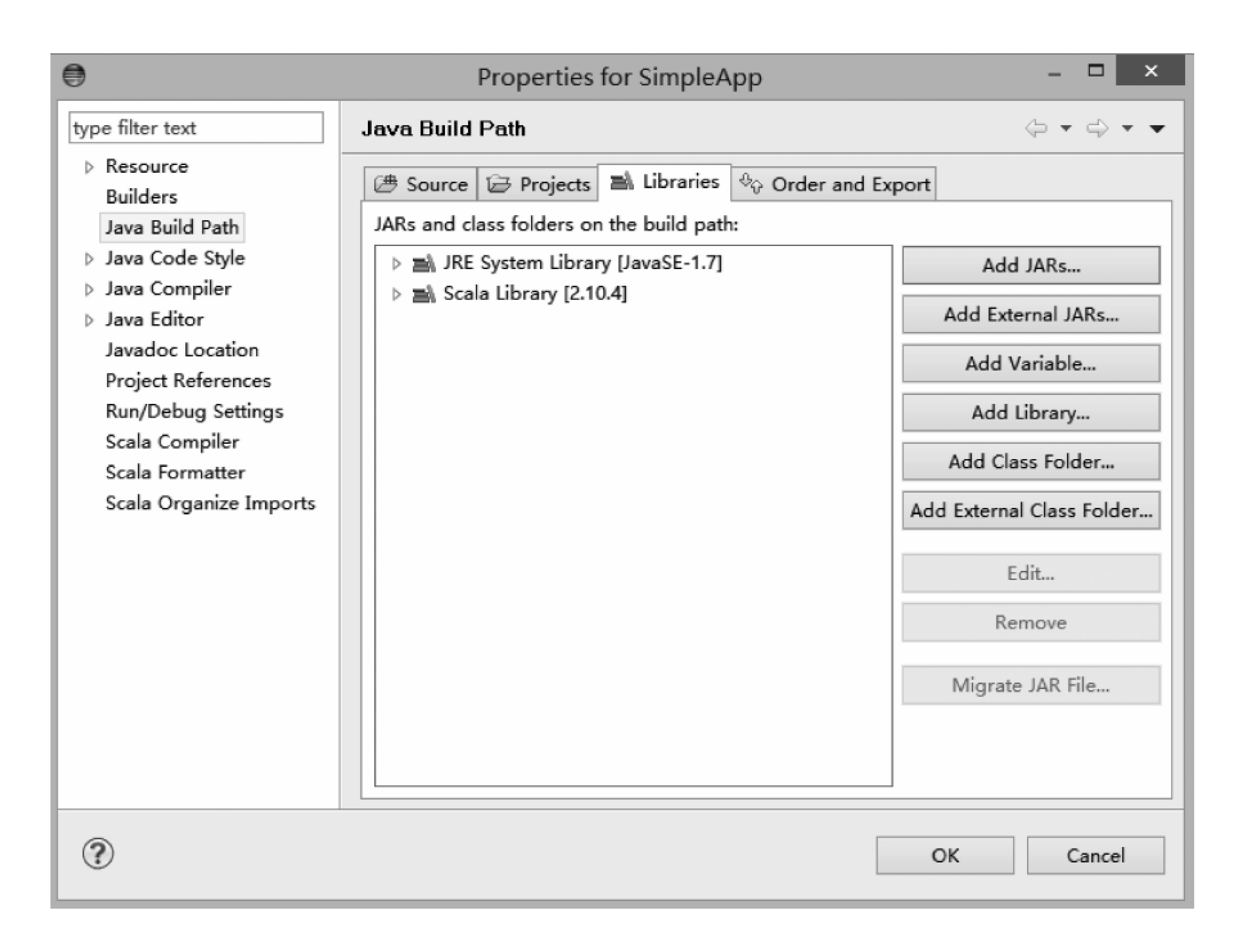


图 2-18 添加 Scala 依赖库

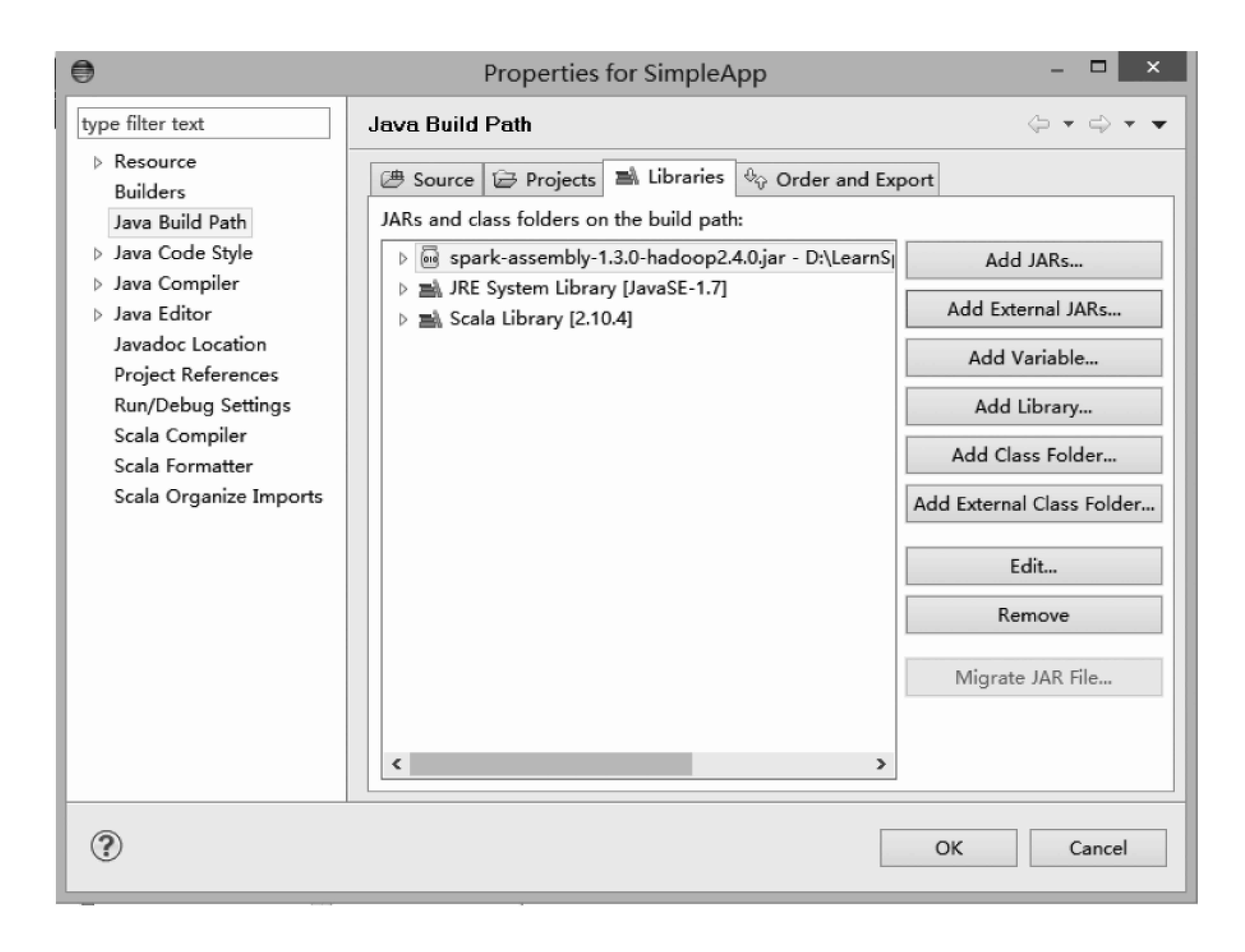


图 2-19 添加 Scala 依赖库成功

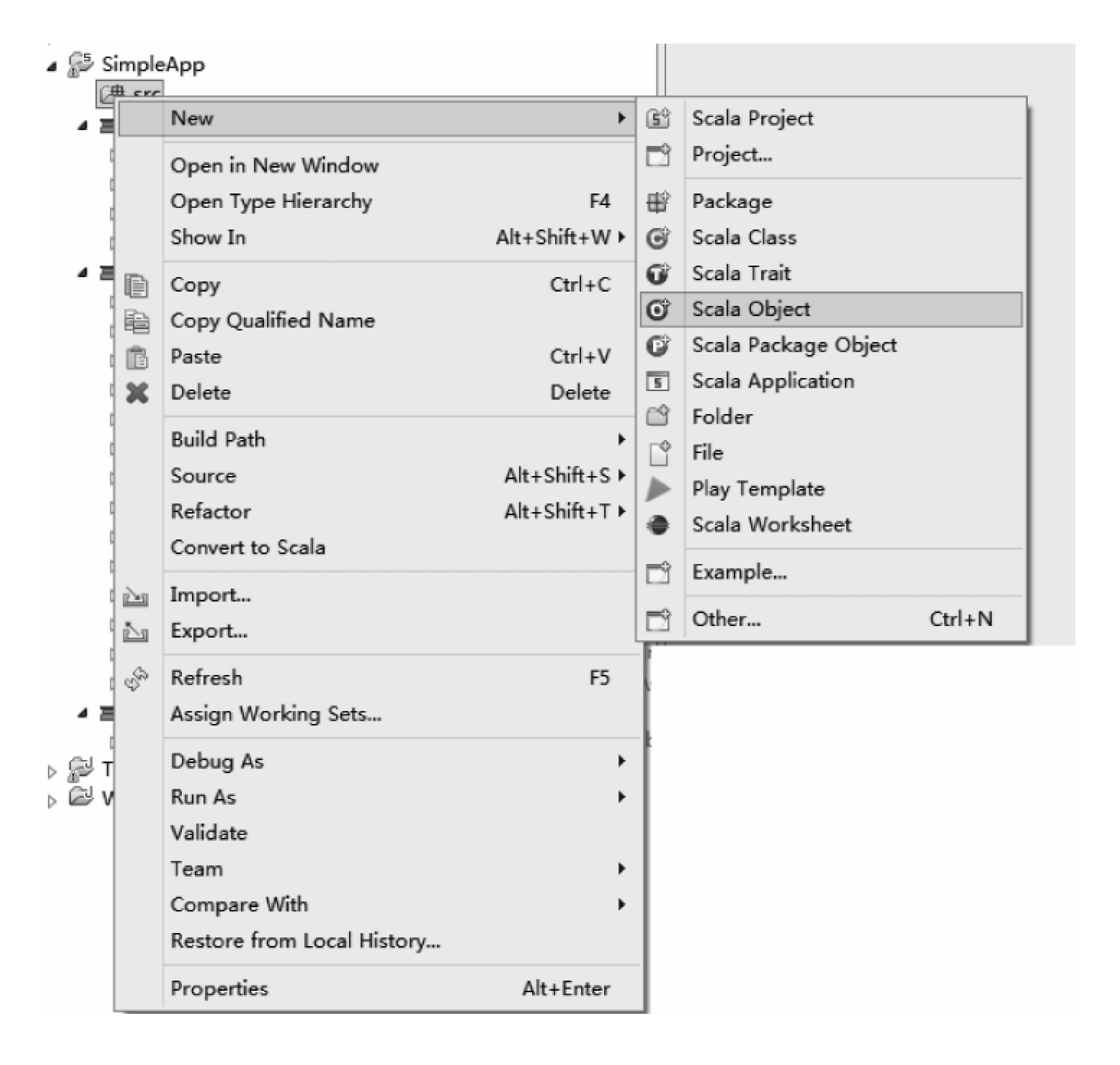


图 2-20 新建 Scala 对象

- 编辑如下的源代码，实现 WordCount 逻辑。

WordCount 程序代码

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object SimpleApp {
  def main(args: Array[String]) {
    if(args.length!=2) {
      println("error : too few arguments")
      sys.exit(1)
    }
    val conf = new SparkConf().setAppName("Simple Application").setMaster("local")
    val filePath=args(0)
    val sc=new SparkContext(conf)
    val file=sc.textFile(filePath, 2).cache()
    val counts=file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
    counts.saveAsTextFile(args(1))
  }
}
```

2.2.3 运行 Spark 程序

为运行示例程序,需要指定两个参数:①输入文件的路径;②输出文件的路径。因此,我们需要按照以下的顺序运行示例程序。

- 选择"Run Configuretions",添加参数,如图 2-21 所示。

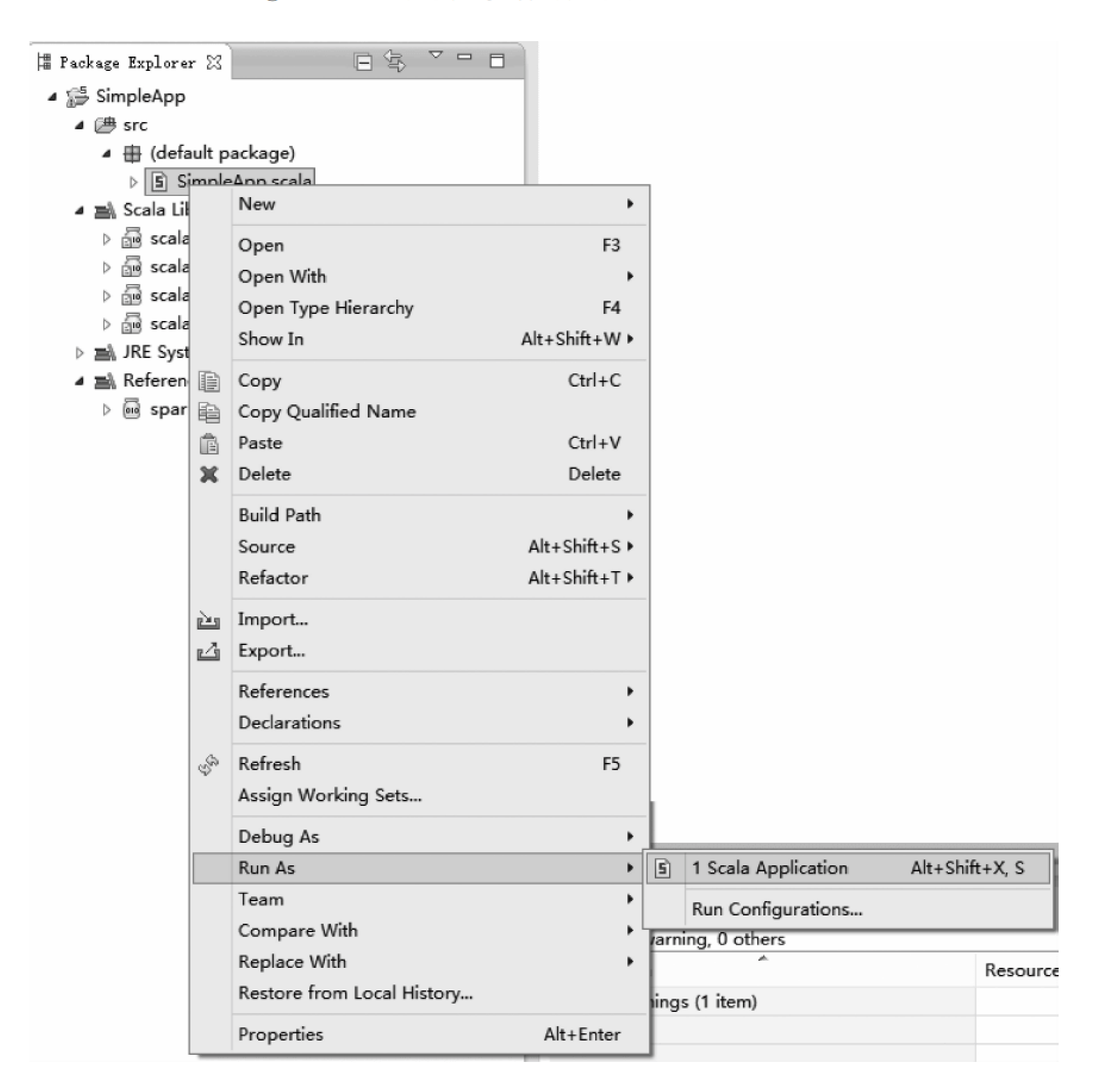


图 2-21 设置程序运行参数

- 在弹出菜单中选择"New"为程序新建运行参数,如图 2-22 所示。
- 设置参数后,点击"Apply"应用这些参数,如图 2-23 所示。
- 配置完成后,即可点击"SimpleApp. scala"文件,使用"Scala Appliction"方式在 Eclipse 中运行该应用,如图 2-24 所示。

在程序运行过程中,Eclipse 会输入如图 2-25 所示的日志,在这里面我们可以看到一些 Spark 作业运行过程中的简要信息,这些过程我们将在后面的章节中具体给大家介绍。在前面的步骤中,我们设置了程序的输出路径是 E:\LearnSpark\count. txt,因

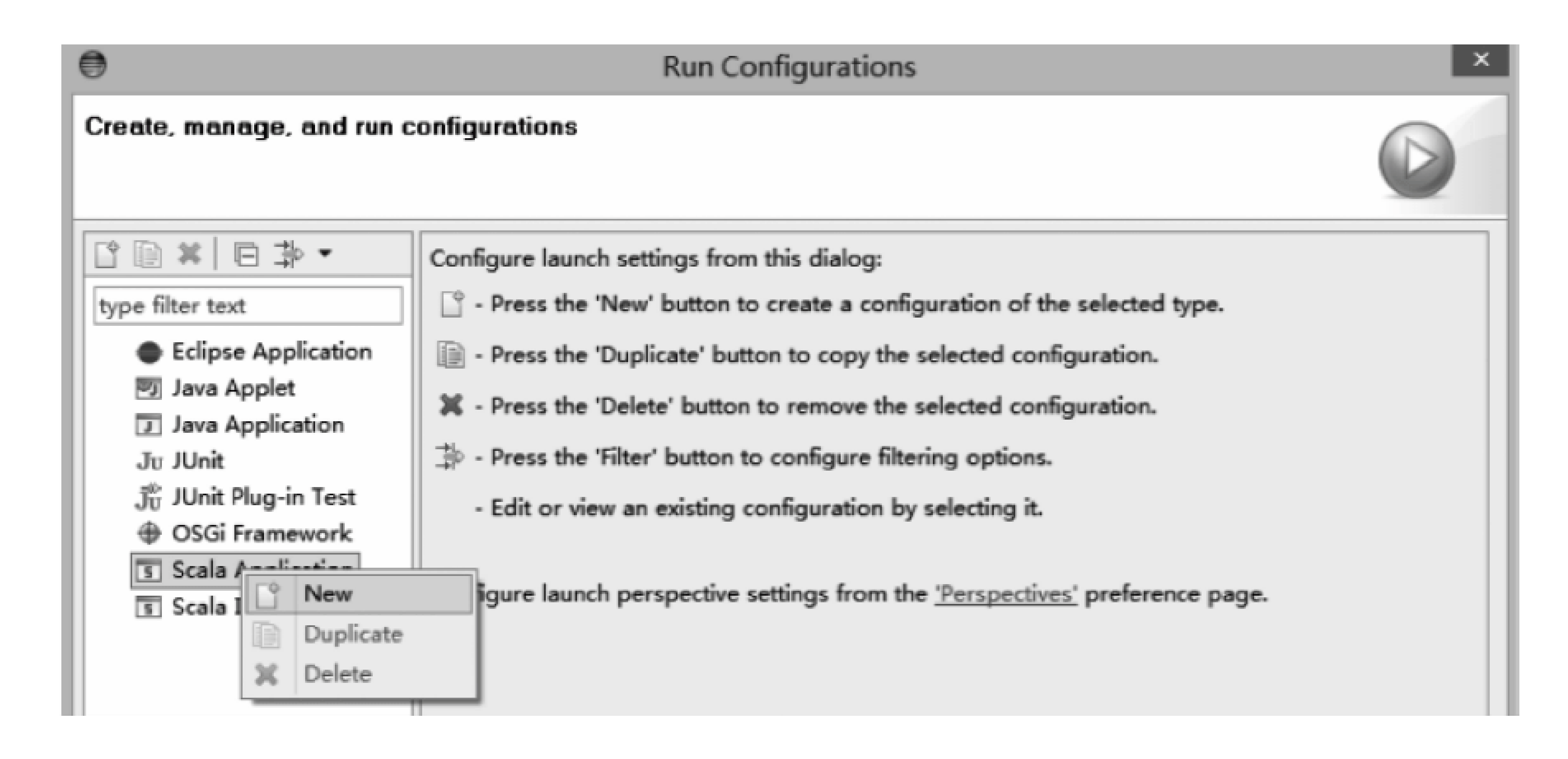


图 2-22 新建运行参数

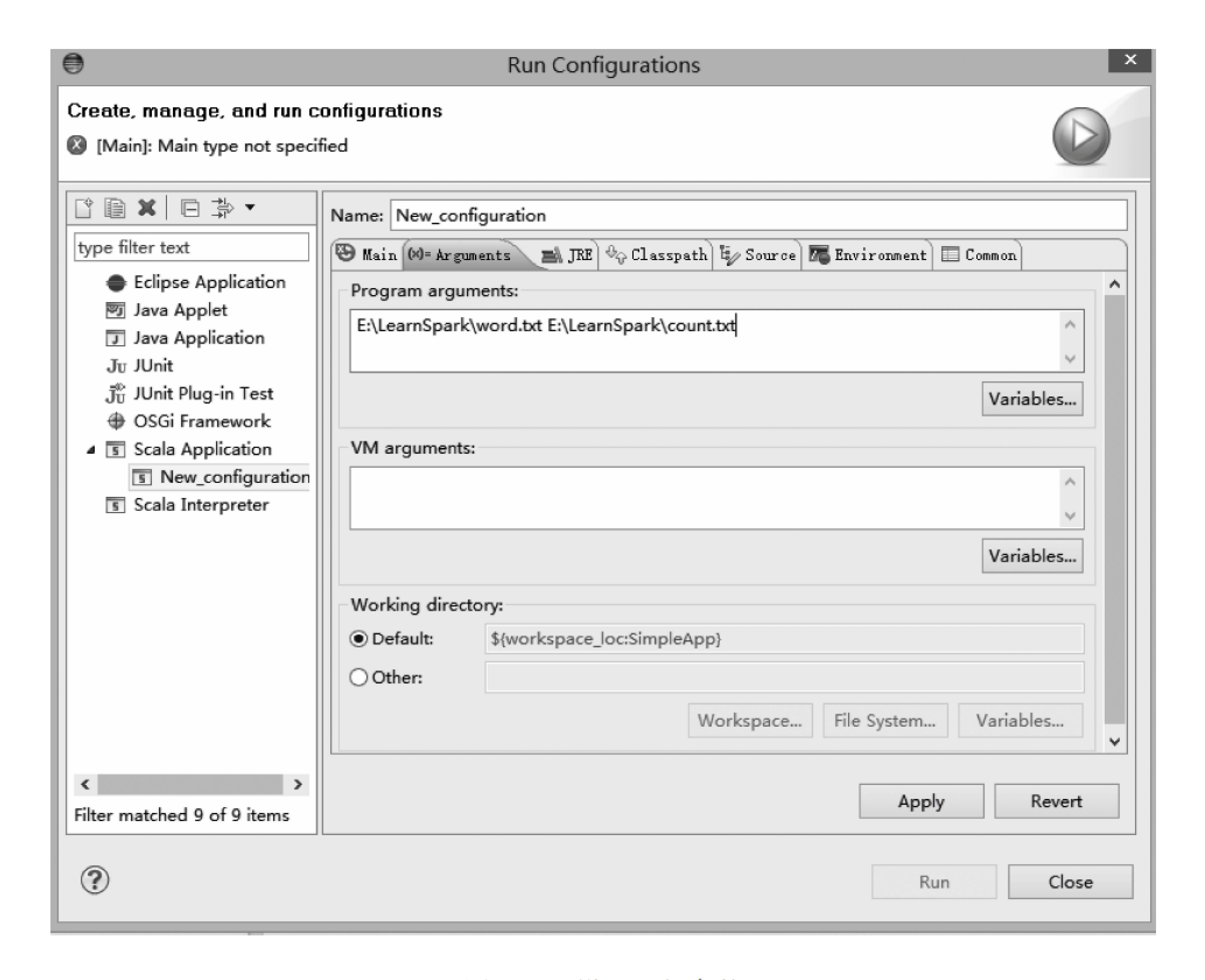


图 2-23 设置运行参数

此，在电脑上打开这个文件就能看到单词计数的结果。到这里，我们就成功地使用 Eclipse 编写了一个 Spark 程序，并在本地环境中运行成功。同样，我们也可以把程序打包为 Jar 包，上传到 Spark 集群中运行，这里我们就不再赘述，有兴趣的读者可以参考 Spark 官方文档详细了解在集群中的运行方法。

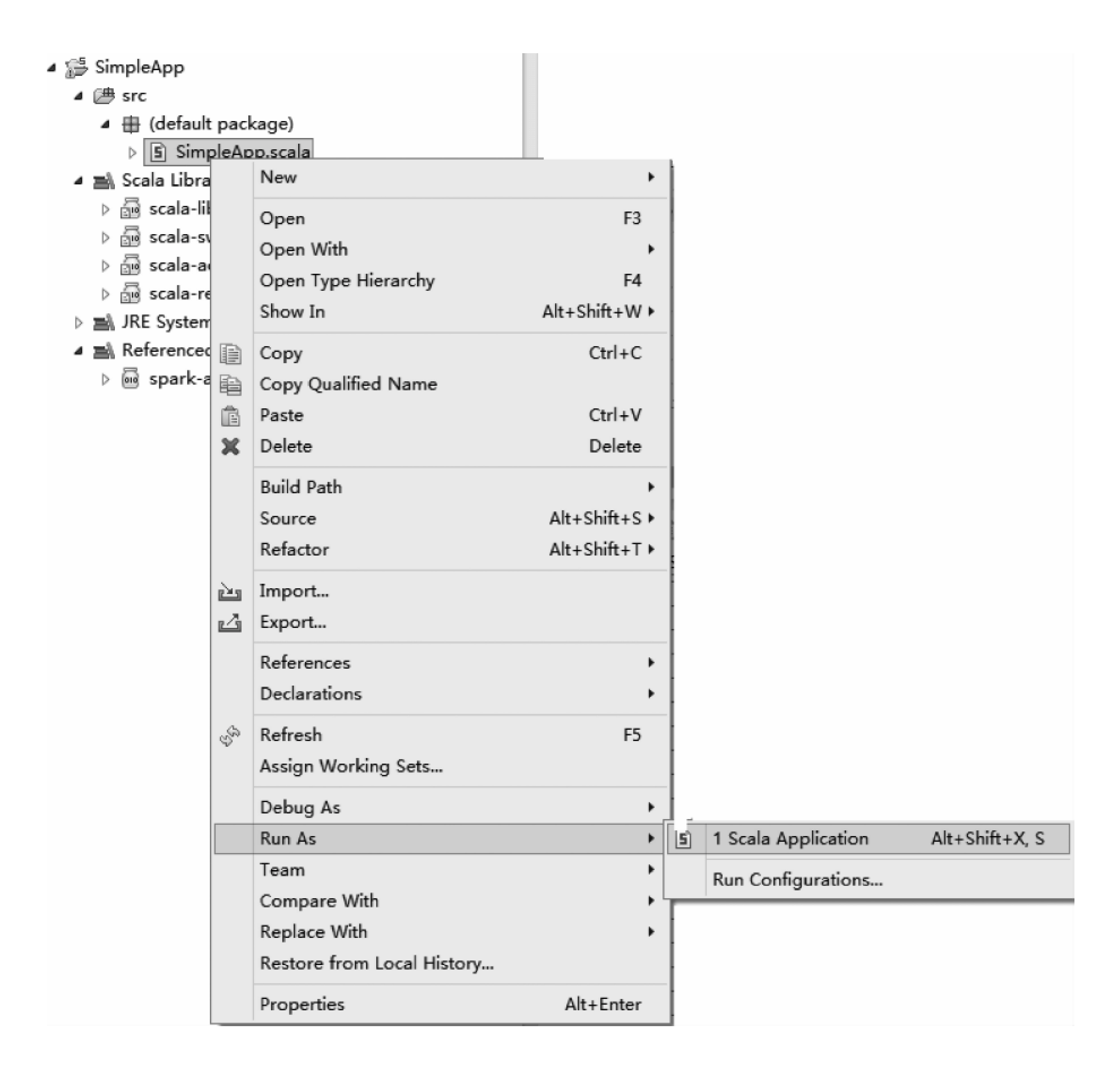


图 2-24 运行示例程序

```
Problems  Tasks  Console
<terminated> SimpleApp$ (1) [Scala Application] C:\Program Files\Java\jre8\bin\javaw.exe (2015年5月11日 上午9:38:38)
15/05/11 09:38:42 INFO TaskSchedulerImpl: Adding task set 1.0 with 2 tasks
15/05/11 09:38:42 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2, localhost, PROCESS_LOCAL, 1056 bytes)
15/05/11 09:38:42 INFO Executor: Running task 0.0 in stage 1.0 (TID 2)
15/05/11 09:38:42 INFO ShuffleBlockFetcherIterator: Getting 1 non-empty blocks out of 2 blocks
15/05/11 09:38:42 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 15 ms
15/05/11 09:38:42 INFO FileOutputCommitter: Saved output of task 'attempt_201505110938_0001_m_000000_2' to file:/D:/LearnSpark/co
15/05/11 09:38:42 INFO SparkHadoopWriter: attempt_201505110938_0001_m_000000_2: Committed
15/05/11 09:38:42 INFO Executor: Finished task 0.0 in stage 1.0 (TID 2). 886 bytes result sent to driver
15/05/11 09:38:42 INFO TaskSetManager: Starting task 1.0 in stage 1.0 (TID 3, localhost, PROCESS_LOCAL, 1056 bytes)
15/05/11 09:38:42 INFO Executor: Running task 1.0 in stage 1.0 (TID 3)
15/05/11 09:38:42 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 2) in 469 ms on localhost (1/2)
15/05/11 09:38:42 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks out of 2 blocks
15/05/11 09:38:42 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 0 ms
15/05/11 09:38:42 INFO FileOutputCommitter: Saved output of task 'attempt_201505110938_0001_m_000001_3' to file:/D:/LearnSpark/co
15/05/11 09:38:42 INFO SparkHadoopWriter: attempt_201505110938_0001_m_000001_3: Committed
15/05/11 09:38:42 INFO Executor: Finished task 1.0 in stage 1.0 (TID 3). 886 bytes result sent to driver
15/05/11 09:38:42 INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 3) in 79 ms on localhost (2/2)
15/05/11 09:38:42 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
15/05/11 09:38:42 INFO DAGScheduler: Stage 1 (saveAsTextFile at SimpleApp.scala:17) finished in 0.516 s
15/05/11 09:38:42 INFO DAGScheduler: Job 0 finished: saveAsTextFile at SimpleApp.scala:17, took 1.118916 s
```

图 2-25 示例程序运行日志

2.3 Spark Web UI

为了便于开发和使用者监控和调试 Spark 应用,Spark 提供了 Web UI 用于查看 Spark 作业在系统中的运行情况。Spark 提供了两种查看作业运行状况的 Web UI,一种是实时 UI,另一种是历史 UI。通过实时 Web UI 可以查看一个 SparkContext 生命周期中运行的 Spark 作业信息,如果某个 SparkContext 生命周期结束了(例如一个 Spark Shell 被关闭),则相关的作业信息对应的实时 UI 就消失了。与此相对地,通过历史 UI 则可以看到所有运行结束后的 Spark 作业信息。由于历史 UI 与实时 UI 的呈现方式基本一致,所以,只需要通过 Spark 的简单配置选项打开历史信息记录和不同的访问端口即可,因此,在本节中我们只对实时 UI 中展示的作业信息进行介绍。

2.3.1 访问实时 Web UI

Spark 的实时 Web UI,依赖于 SparkContext 对象实例创建后启动的 Web 服务,因此,我们首先要启动一个 Spark 程序以触发创建 SparkContext 对象实例。例如,我们可以通过启动 Spark Shell 命令的方式创建一个 SparkContext 对象实例。下面我们在命令行窗口输入(这里假定你已经完成了 2.1 节的全部操作和配置):

```
> bin \spark - shell
```

Spark Shell 环境启动后,随之启动了一个 Web 服务,查看实时作业信息的 Web 服务默认在 4040 端口监听,因此,我们在浏览器的地址栏输入以下地址即可访问实时 Web UI 界面: http:// localhost:4040。

因为我们的演示是在单机环境下进行,因此,上面的 URL 中服务器地址为 localhost。如果在集群环境下运行,更换为 Spark 作业的 Driver 节点的 IP 地址即可。当然,我们也可以在一台服务器上启动多个 Spark 程序,每启动一个,就会开启一个新的 Web 服务显示该程序的实时信息,访问这些 Web 服务的方式只需要将地址栏输入的端口号由 4041 依次往后加 1 即可。

在仅启动 Spark Shell 环境而没有运行 Scala 脚本、运行 Spark 程序的情况下,实时 Web UI 中是没有任何作业信息的,其显示如图 2-26 所示。

从图中顶部菜单,我们可以看到,实时 Web UI 可以显示 Spark 作业的以下信息。

- Jobs: 显示的是作业相关信息,已经完成的作业有哪些,正在进行的作业有哪些等。

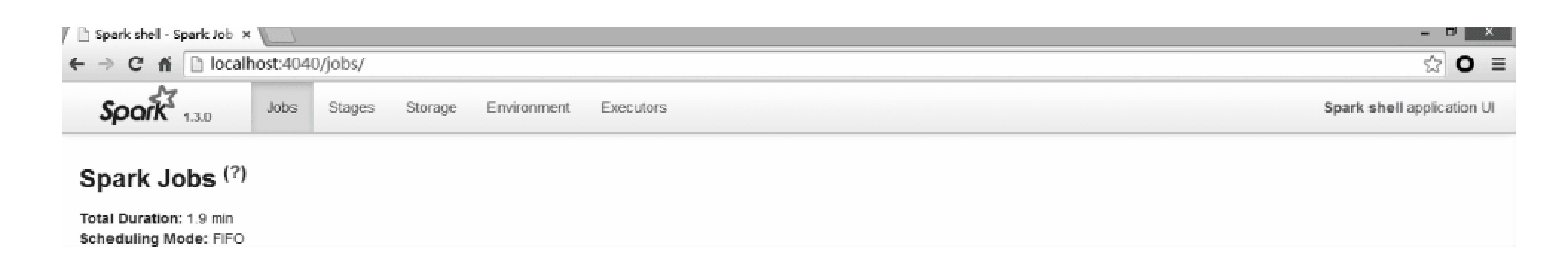


图2-26 Spark Web UI 默认页面

- Stages：在 Spark 中，一个作业会被划分为多个阶段（Stage）执行，Stages 页面显示的是作业各阶段的信息。
- Storage：显示的是 RDD 存储情况，包括存储的方式及大小。
- Environment：显示 Spark 运行的相关环境变量及各种参数，例如 JAVA_HOME 等。
- Executors：显示 Spark 作业并行执行时每个执行器（Executor）的情况。

在菜单的下方还有两个重要信息：一个是当前 SparkContext 存活的时间，这里显示的是 Spark Shell 被启动了 1.9 分钟；另一个是目前 Spark 程序的调度模式。Spark 的调度模式在后面的章节中我们会介绍，这里我们只需要知道当前作业采用的是先入先出（FIFO）调度机制。

2.3.2 从实时 UI 查看作业信息

为了在实时 UI 中显示出作业信息，我们在打开的 Spark Shell 窗口输入以下语句执行：

```
scala> val file = sc.textFile("E:\\LearnSpark\\word.txt")
scala> file.cache
scala> val counts = file.flatMap(line => line.split(" ")).map(word =>
(word, 1)).reduceByKey(_ + _)
```

按照一个正常的程序逻辑，在输入以上语句后，程序应该是要执行了。但此时你可以切换到实时 Web UI 页面，会发现看不到任何作业运行的信息，这就是 Spark 程序的惰性（Lazy）执行特性。只有当我们再输入以下语句时，这个 Spark 作业才真正开始执行。

```
scala> counts.saveAsTextFile("E:\\LearnSpark\\counts.txt")
```

此时，再打开实时 Web UI 页面，我们就能看到以上语句触发执行的 Spark 作业信息。

- **作业信息页面（Jobs）**

如图2-27所示，在作业信息页面，我们能看到已经有一个完成的作业。由于这个

作业计算量很小,因此,在提交后 0.8 秒就完成了。该作业分为了两个阶段(Stage)执行,均成功完成。一共执行了 4 个作业,并且也都成功完成。如果提交的是计算量较大的作业,在 Tasks 列可以看到作业执行的进度。

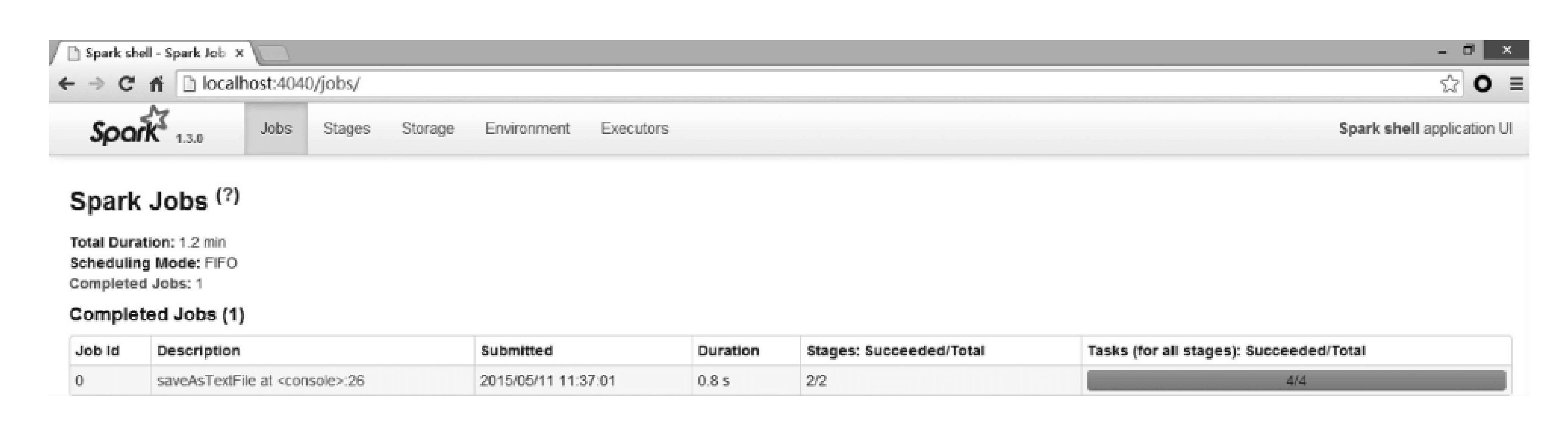


图 2-27 作业信息页面

- **作业阶段信息(Stages)**

如图 2-28 所示,我们可以看到这个作业被切割成了两个阶段执行,并且均已执行完成。其中阶段 1(Stage Id 为 0)执行了 0.4 秒,成功执行了两个任务(Tasks),读入(Input)了 27 个字节数据,输出了 509 个字节的 Shuffle 数据(Shuffle Write)。阶段 2(Stage Id 为 1)执行了 0.3 秒,成功执行了两个任务(Tasks),从上一阶段(阶段 1)读取了 509 个字节的 Shuffle 数据(Shuffle Read)。点击 Stage 的描述(Description),还可以查看 Stage 执行的详细信息,在这里我们暂时不进一步深入,留待后续介绍 Spark 原理时再具体展开。



图 2-28 作业阶段信息页面

- **存储信息页面(Storage)**

在上面的代码中,为了触发 RDD 进行存储,我们执行了一个语句“file.cache”,因此,可以在存储信息页面看到该 RDD 存储的情况,如图 2-29 所示。这个 RDD 的名称是 E:\LearnSpark\word.txt,即我们读入的文件。该 RDD 被存储在内存(Memory)中,分为两个分区(Partitions),在内存中存储了 184 个字节,在 Tachyon(Spark 使用的一种内存文件系统)和磁盘没有存储。点击 RDD 的名称还可以查看 RDD 存储更为详细的

信息，在这里我们也暂时不作展开。

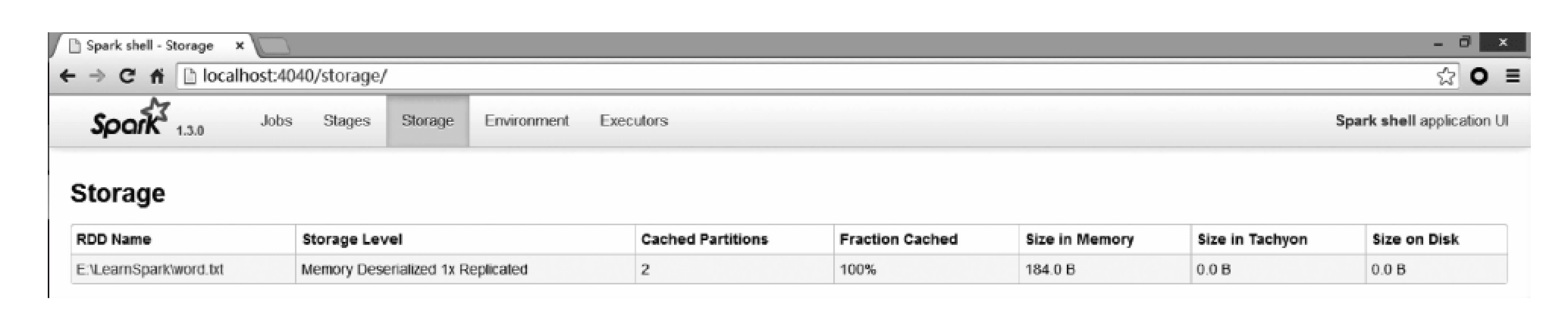


图 2-29　存储信息页面

- **环境变量页面（Environment）**

Spark 作业执行中的各种环境变量、配置参数等都可以在该页面方便地查看。例如，可以查看到基本的运行配置（例如 Java Home、Java Version），还可以看到 Spark 相关的端口（例如 spark. driver. port）等重要信息，如图 2-30 所示。

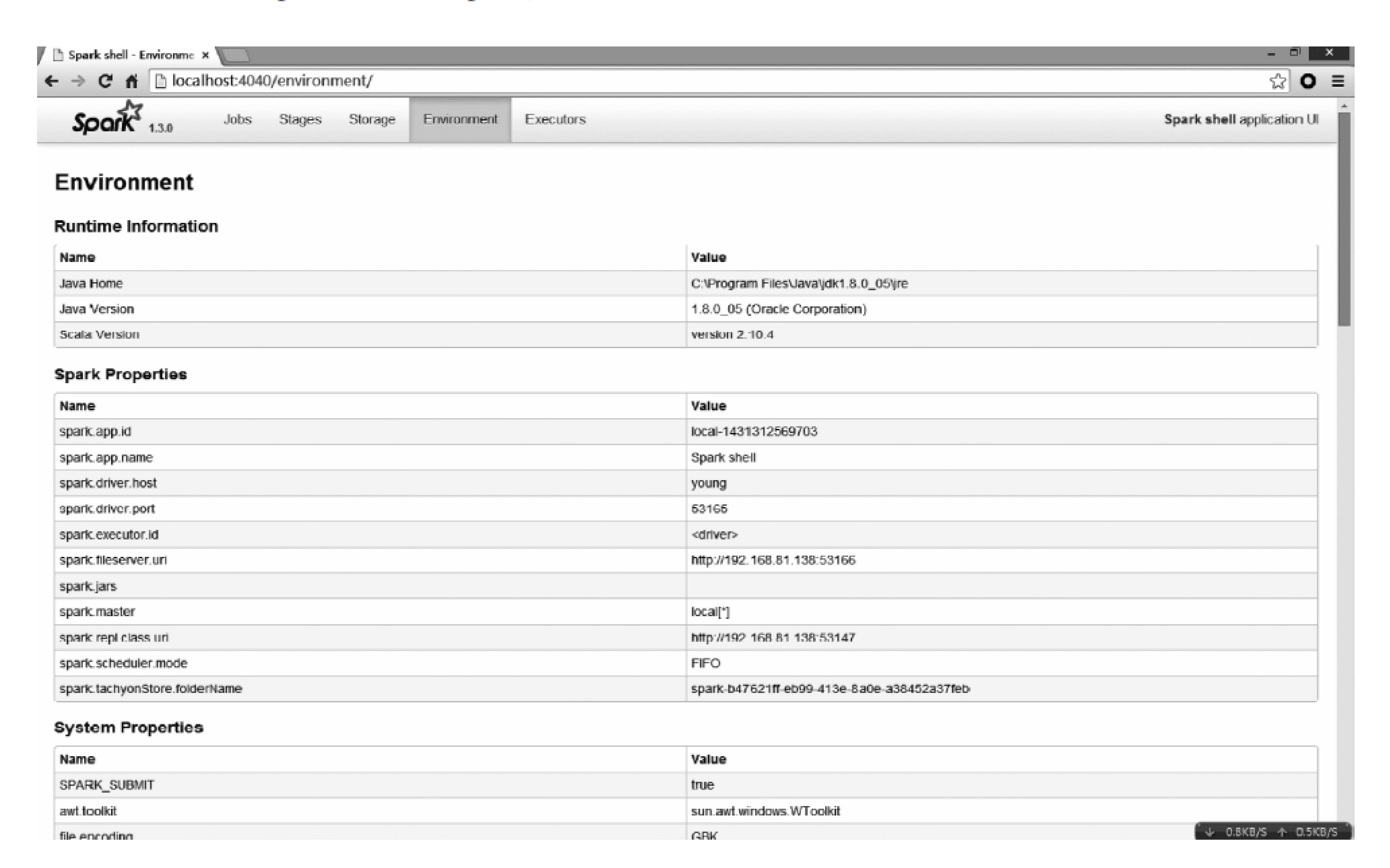


图 2-30　环境变量信息页面

- **执行器信息页面（Executors）**

Spark 作业可以由多个执行器（Executor）并行执行，由于我们的试验是在单机环境下运行的，因此，我们看到的是只有一个 Driver。该 Driver 在本机（localhost）的 53 178 端口进行监听，生成了两个 RDD 块，没有使用内存和磁盘，已经成功完成了 4 个任务，执行了 1.1 秒，读入了 27 个字节数据，输出了 509 个字节的 Shuffle 数据，如图 2-31 所示。

至此，我们基本了解了使用 Web 界面查看 Spark 作业运行情况的方法。初步感受

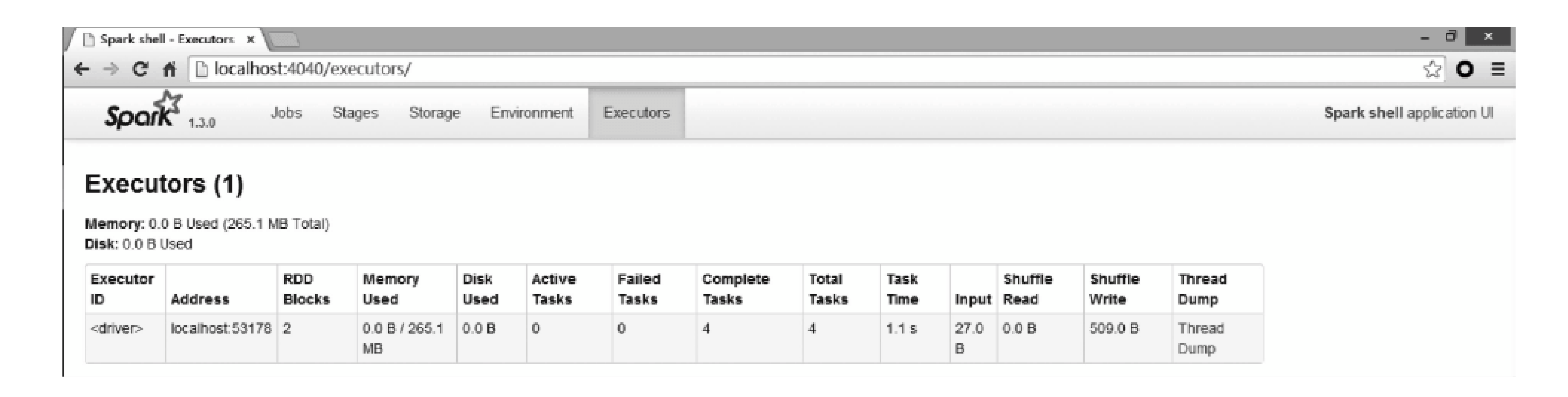


图 2-31　执行器信息页面

到了 Spark 的几个特性：①Spark 并行运行时会分成 Driver 和 Executor；②Spark 在执行作业的时候会把一个作业根据某些策略划分成不同的阶段(Stage)；③Spark 中有些算子会提交作业处理数据，有些算子并不提交作业来处理数据；④Spark 可以把一些需要处理的数据缓存在内存中，或者是存储在以 Tachyon 为代表的分布式内存文件系统。在后面的章节中，我们会结合这些页面的信息继续深入探讨 Spark 的原理和运行机制，以及使用这些页面中的信息帮助我们优化和改进 Spark 程序。

第3章

Spark 原理

在前面的章节中，我们已经体验到了与 Spark 相比 Hadoop 的性能优越性和使用便捷性。当然，Spark 的这些优势并不是凭空产生的。作为大数据处理技术的后来者，Spark 的设计目的就是在已有技术的基础上继续降低编写数据处理程序的难度和提高数据处理的效率。因此，Spark 吸收和借鉴了包括 MapReduce 计算模式、函数式编程思想、DAG[10] 执行模式等多种已有技术的优点，并巧妙地融合于基于弹性分布式数据集实现的计算框架中，最后形成了一个简洁高效的创新分布式处理框架。为了更好地掌握和使用这一框架，下面我们就以大家最熟悉的 WordCount 程序为例，初步深入 Spark 的原理和运行机制，为后面的 Spark 算法设计奠定基础。

3.1 Spark 工作原理

在开始了解 Spark 工作原理之前，我们先介绍 Spark 中两个最重要的基本概念：弹性分布式数据集（Resilient Distributed Datasets，RDD）[11] 和算子（Operation）。

- **RDD**

Spark 使用弹性分布式数据集 RDD（Resilient Distributed Datasets）实现数据存储。RDD 可以理解为将一个大的数据集合以分布式的形式保存在集群服务器的内存中。从物理上来讲，RDD 将一个大的数据集分块为一系列数组，这些数组以分布式的方式存储在集群中的各个节点上，每个数据块有一个标识，称为 BlockID。这样就可以通过记录每个分块的元数据（Meta Data）对数据块进行管理。每个数据块可以存储在节点的内存中，或者被持久化在硬盘上。为了便于组织和处理，这些数据块在逻辑上进行划分后，形成多个分区（Partition）。用户可以通过对 RDD 使用一系列变换算子的处理，并利用行动算子触发实际的操作。Spark 根据相应的变换计算流程，对输入 RDD 的分区数据计算得到新的结果 RDD。这样一来，用户就能够以类似编写单机程序的方

法来进行分布式计算。

- **算子**

算子是 Spark 中定义的函数，用于对 RDD 中的数据进行操作和转换。Spark 中的算子可以分为 4 类。

（1）创建算子（Creation），用于将内存中的集合或外部文件创建为 RDD 对象。

（2）变换算子（Transformation），用于将一个 RDD 转换为另一个 RDD。

（3）缓存算子（Cache），用于将 RDD 缓存在磁盘或内存中，以便后续计算重复使用。

（4）行动算子（Action），会触发 Spark 作业执行，并将计算结果 RDD 保存为 Scala 集合或标量，或者保存到外部文件或数据库中。

第一次接触 Spark 的读者，可能对以上两个概念会感觉比较晦涩。我们仍以大家已经熟悉的、非常简单的 WordCount 程序为例，结合 WordCount 程序的每一步计算来理解以上两个重要概念和 Spark 的工作原理。下面是用 Spark 实现 WordCount 程序的代码。

基于 Spark 的 WordCount 程序

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object SimpleApp {
  def main(args: Array[String]) {
    if(args.length != 2) {
      println("error : too few arguments")
      sys.exit(1)
    }
    val inputFile = args(0) //读取输入文件路径参数
    val outputFile = args(1) //读取输出文件路径参数
    val conf = new SparkConf().setAppName("WordCount").setMaster("local") //生成配置实例
    val sc = new SparkContext(conf) //生成 SparkContext 实例
    val file = sc.textFile(inputFile, 3) //读取输入文件内容生成 RDD
    val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _) //统计
    counts.saveAsTextFile(outputFile) //保存统计结果
  }
}
```

我们可以将整个程序分成两个部分。

（1）程序代码的前 12 行都是进行 Spark 程序的前期准备工作。代码第 1 ~4 行引入必要的包。第 5 ~9 行验证程序运行必要的两个参数是否具备，第 1 个参数是输入文件的路径，第 2 个参数是计数结果文件的输出路径。第 10 行是从第 1 个参数中取出输入文件路径。第 11 行是从第 2 个参数中取出输出文件路径。第 12 行是创建 Spark 程序运行需要的配置实例，其中设置应用的名称为“WordCount”，并采用本地方式运行。

（2）从代码第 13 行到程序结束是执行单词计数逻辑的代码。第 13 行是创建运行 Spark 程序必需的 SparkContext 实例。SparkContext 是整个 Spark 程序的入口，负责向集群申请程序运行的环境和资源以及进行必要的初始化工作。一旦资源初始化完成、集群上资源申请成功，程序就可以进行对数据的计算操作。第 14 ~ 16 行是执行 WordCount 算法的主要代码。相比 MapReduce 版的 WordCount 程序（具体可参见《Hadoop 大数据处理》一书），你可以看到使用 Spark 实现 WordCount 程序非常简单。实现数据输入、WordCount 计算、数据输出 3 个功能只需要 3 行代码。其中，第 15 行代码是 WordCount 计算的主体，使用了 Scala 的函数式编程方法并调用 Spark 的多个算子。Scala 的函数式编程方法最大的特点就是每次操作都是一次方法调用过程。首先调用 flatMap 算子将输入文件的每一行数据以空格分割为独立的单词，然后调用 map 算子给每一个独立单词计数为 1，最后调用 reduceByKey 算子将内容相同的单词累加求和。

为了更清晰地了解 Spark 的基本原理，我们用图 3-1 展示了上述第 13 ~16 行代码实现的 WordCount 逻辑在 Spark 上的运行过程。

（1）图中第 1 步执行的是代码第 14 行。textFile 是一个创建算子（我们将在下一章详细介绍各类算子），其功能是从 inputFile 指定的 HDFS 文件中逐行读取文本数据，并转化为 Spark 中的 HadoopRDD 对象实例。“textFile”算子中的参数“3”是指生成的 HadoopRDD 数据实例，以 3 个分区（Partition）的方式进行存储，Spark 将 RDD 的分区以分布式的形式存储在集群中不同的物理服务器中。在图中第一步后，我们可以看到 HadoopRDD 的数据被分成 Partition1、2、3 存储，每个分区的每条内容对应输入文件的一行文本信息，在本例中是包含以空白符分隔的若干表示水果名称的单词。

（2）Spark 中的分区只是一个逻辑上的概念，在 Spark 集群上数据的真正存储单元被称为数据块（Block），由数据块管理器（BlockManager）维护和管理。逻辑概念上的分区和物理上的数据块一一对应，即一个分区对应着一个数据块。每个 Block 都有一个

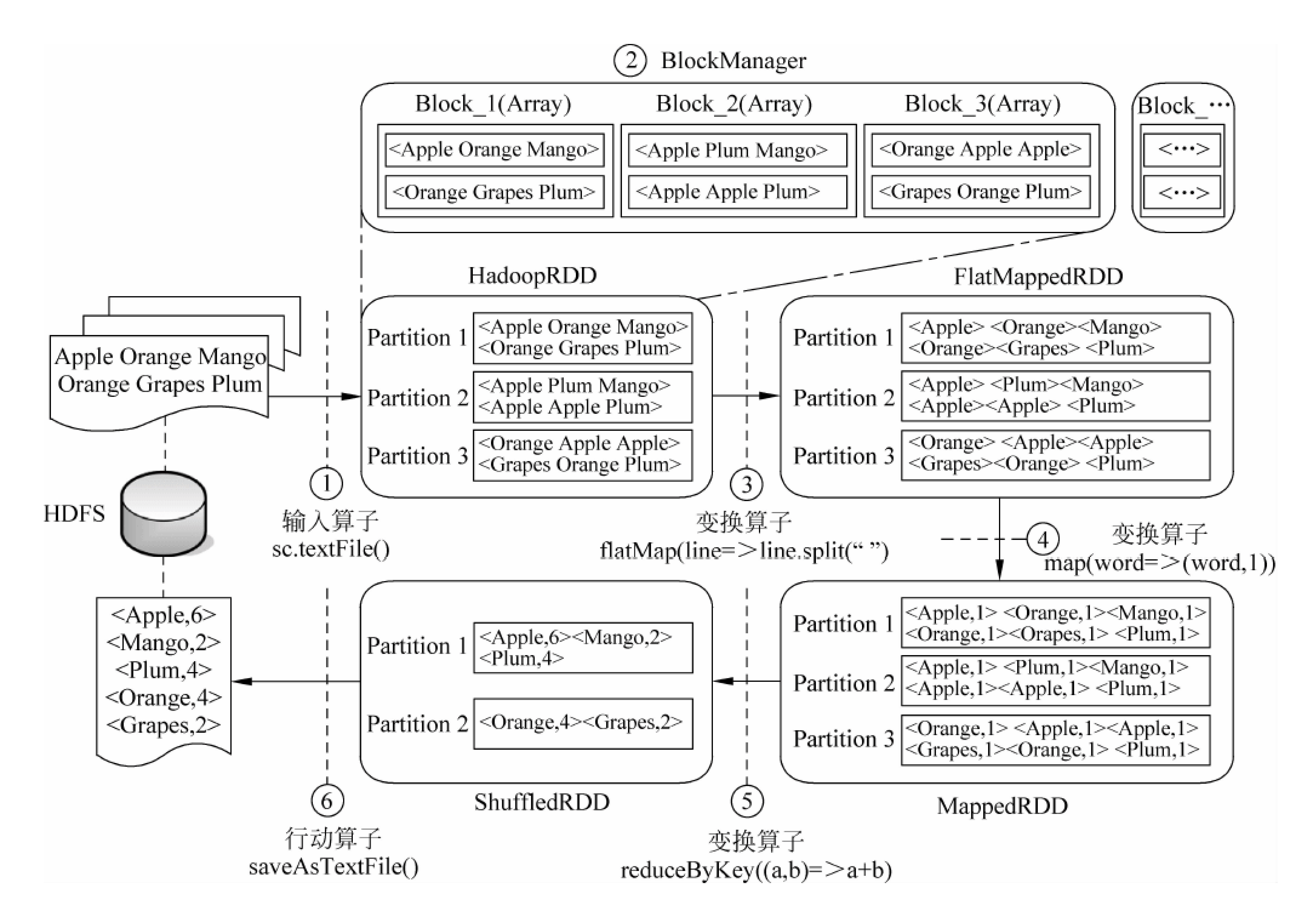


图 3-1 WordCount 运行过程

唯一的 BlockID，其构成方式是"rdd_" + rddID + "_" + partitionID，其中 rddID 是该 Block 所属的 RDD 的标识，partitionID 是对应的 Partition 的标识。BlockManager 依据 BlockID 记录每个数据块的存储节点位置等元数据，以对这些 Block 进行管理。

（3）图中第 3 步对应代码第 15 行的第一部分 file. flatMap( )。flatMap 是一个变换算子，其作用是将原始 RDD 中的每个元素用一个指定函数进行一对多的变换，然后将变换后的结果汇聚生成新的 RDD。在我们的这个例子中，指定函数是 line = > line. split(" ")，其功能是将原始 RDD 中的每一行文本，按空格为分隔符，分隔成单词。因此，第 3 步的作用就是将从文件中读入的数据，按空格分隔拆分为一个一个的单词，生成一个新的 RDD，我们称为 FlatMappedRDD。由于 flatMap 算子是对原始 RDD 的每个 Partition 进行一对一的变换，因此，生成的新 RDD 仍然为 3 个 Parition。

（4）图中第 4 步对应代码第 15 行的第二部分 map(word = > (word, 1))。map 也是一个变换算子，其功能是将一个指定函数作用于原始 RDD 的每个元素，形成一个新的 RDD。在我们的这个例子中，指定函数是 word = > (word, 1)，其作用是对每个单词进行变换，生成一个单词为 Key，频数 1 为 Value 的键值对(Key/ Value Pair)。所有变换生成的键值对构成一个新的 RDD，我们称之为 MappedRDD。同样，Map 算子也是

对原始 RDD 的每个 Partition 进行一对一的变换,因此,生成的 MappedRDD 仍然为 3 个 Parition。

(5) 图中第 5 步对应代码第 15 行的第三部分 reduceByKey(_ + _)。reduceByKey 仍然是一个变换算子,但跟前面 flatMap 和 map 算子不同,reduceByKey 算子的作用对象是 Key/ Value 型的 RDD。reduceByKey 算子将 Key/ Value 型 RDD 中 Key 值相同的 Value 值按照参数指定函数进行归并,然后生成新的 RDD。在我们的这个例子中,指定函数是"_ + _",这是 Scala 语法中的一个特殊写法,逻辑是将所有 Value 值逐个相加,得到对应 Key 值(也就是单词)的出现次数。reduceByKey 算子对全部分区进行归并计算,所以,生成的新 RDD 就不是原来的 3 个了,而是 Spark 集群的默认 Partition 数。我们这里假定默认 Partition 数的参数为 2,因此,生成的 ShuffledRDD 中只有 2 个分区。

(6) 经过第 3、4、5 步,我们已经统计出了输入数据中每个单词的出现次数,并存放在 ShuffledRDD 中。为了输出结果,在代码的第 16 行,我们使用行动算子 saveAsTextFile 将 ShuffledRDD 中的数据保存到 HDFS 文件中。至此,统计单词频数的 WordCount 程序完成。

通过上面的示例,我们基本了解了 Spark 通过 RDD 和算子两个基本原理实现分布式并行计算的原理。在这个过程中,我们需要记住 Spark 中的两个重要理念。

(1) 在 Spark 中,只支持对 RDD 进行粗粒度的操作,即只能对 RDD 进行整体性的操作,而不提供对 RDD 中某个元素进行操作的函数接口。例如,在 WordCount 程序中,我们都是对由句子、单词构成的整体 RDD 数据集进行变换,而没有哪一行代码对某一个句子或单词进行操作。

(2) 在 Spark 中,RDD 是只读的。上面的每次变换都是由一个 RDD 生成新的 RDD 的过程。这一系列 RDD 的变换操作过程构成了一个操作序列,以达到最终的计算目的。

从上面的代码和原理讲解中我们可以看到,使用 Spark 实现 WordCount 功能是如此简单。显然,Spark 计算框架在后台默默为我们完成了大量的支持工作才能使我们编写大数据处理程序如此简单。那么,Spark 究竟为我们做了什么呢? 在下一节我们仍然以 Spark 版本的 WordCount 程序为例,来看一看 Spark 程序的执行过程。

3.2 Spark 架构及运行机制

3.2.1 Spark 系统架构与节点角色

与 Hadoop 技术族中的 MapReduce 和 HDFS 类似,Spark 也采用了相对成熟的主从

(Master/ Slave)架构构建计算集群,其架构如图 3-2 所示。其中 Client 为提交 Spark 程序的节点。其余为 Spark 分布式集群中的物理节点,这些节点可以分为两类,集群管理(ClusterMaster)节点和从(Slave)节点。

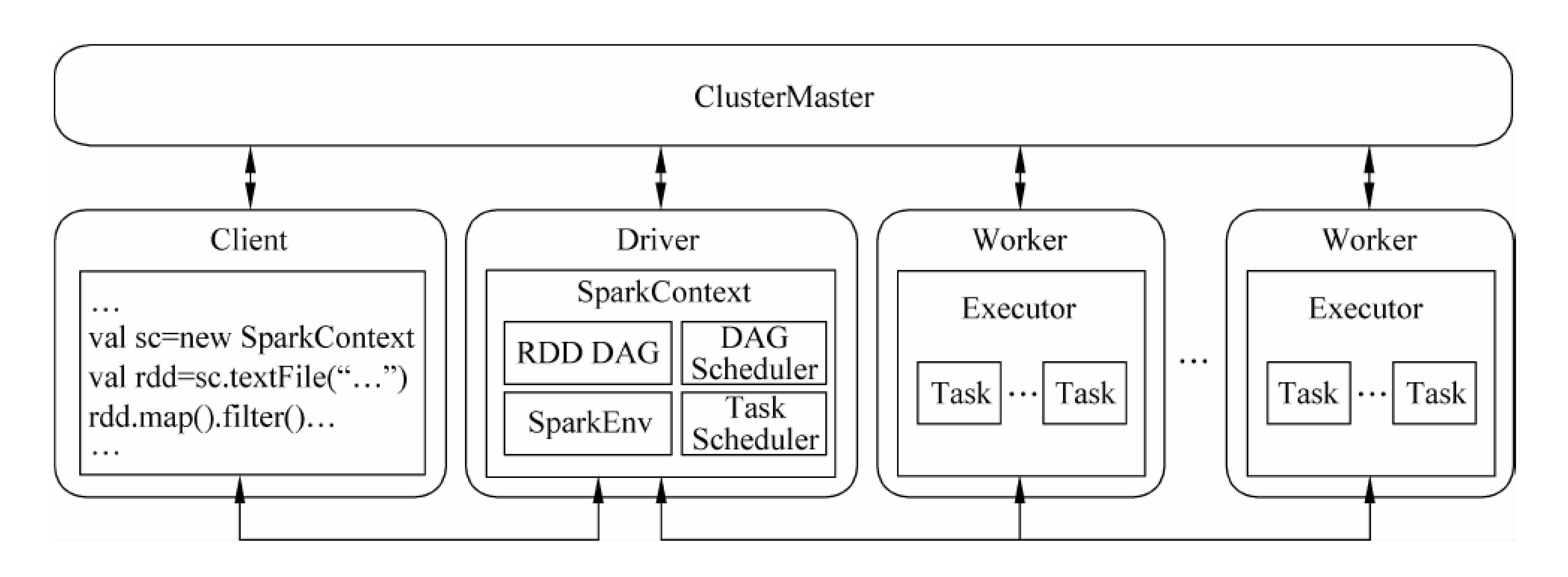


图 3-2 Spark 系统架构

- ClusterMaster 节点:ClusterMaster 是整个 Spark 集群的核心,在集群中所处的地位与 HDFS 集群中 NameNode 节点的地位类似。ClusterMaster 节点并不执行实际的计算任务,而是负责管理整个集群的计算资源,这里所说的计算资源是指除 ClusterMaster 节点外其他物理主机的内存、CPU 处理器等物理资源。这些计算资源都由 ClusterMaster 节点进行统一管理,并将资源合理地分配给用户提交的各个应用程序。所有计算节点都要向 ClusterMaster 节点进行注册,将自身的计算资源交给 ClusterMaster 节点进行统一调度。ClusterMaster 节点随时监控了解这些注册的节点的运行状况,以便为应用程序提供合理的资源分配。需要注意的是,ClusterMaster 节点是一个逻辑上的概念,当 Spark 集群采用不同模式运行时,ClusterMaster 节点对应的是这些模式中的相应管理节点。例如,以简单的 Standalone 模式运行 Spark 时,ClusterMaster 节点就是运行 Master 服务的节点。以 YARN 模式运行 Spark 时,对应的 ClusterMaster 节点为 YARN 中的 ResourceManager 节点。而以 Mesos 模式运行 Spark 时,对应的 ClusterMaster 节点则为 Mesos 中的 Master 节点。
- Slave 节点:Slave 是 Spark 集群中执行作业逻辑的节点。但与 HDFS 集群中作为 Slave 节点的 DataNode 节点只完成单一功能不同,Spark 中的 Slave 节点又根据功能不同分为两类:任务调度节点(Driver)和任务执行节点(Worker)。区分这两种节点的根本方法就是看 Slave 节点上运行着哪种功能的进程。
  √ Driver 节点:如果节点上运行了 Spark 程序 main 函数所在的进程,那么该节点就作为该应用程序的 Driver 节点。在 Spark 集群中,Driver 进程可以运行

在提交 Spark 程序的 Client 节点上，也可以是某个 Worker 节点上。例如，当我们用前面示例中运行 Spark Shell 时，实际上启动了一个 Spark 程序，因此，我们的电脑就是 Driver 节点。Driver 节点作为整个应用程序逻辑的起点，负责创建 SparkContext、定义一个或者多个 RDD，同时 Driver 进程还可以看做是整个应用程序的大脑。Driver 主要负责两方面的工作：①负责将一个应用程序分割为物理上可执行的任务（Task），在 Spark 中，Task 是物理上可执行的最小单元，一个应用程序可以启动成百上千的独立 Task；②对于应用程序产生的 Task，Driver 进程将 Task 任务分配到最适合的 Worker 节点上运行，并协调这些 Task 在 Worker 上完成运行。这些过程我们会在后面的内容中详细介绍。

√ Worker 节点：运行 Executor 进程的节点为 Worker 节点，Spark 为每一应用程序在 Worker 节点上创建一个 Executor 进程，Executor 进程是实际物理任务的执行者。Executor 进程负责两方面的工作：①负责执行组成应用程序的独立 Task 计算任务，并将执行的结果反馈给 Driver 节点；②Executor 进程为 RDD 提供内存存储。

上面的 Spark 功能节点，可以看作执行 Spark 程序的不同功能进程，这些进程可以用不同模式运行在一个或多个物理节点中。因此，Spark 程序可以在单机上运行（例如我们在前面的示例中以 Spark Shell 形式运行的 Scala 脚本），也可以在集群中分布式运行。而且，在集群中运行时，也可以采用多种资源调度框架。这些差异，就构成了 Spark 程序可以运行的多种模式，具体使用哪种模式可以通过设置和程序传递到 SparkContext 的 MASTER 环境变量值确定。这些模式包括：

（1）本地（local）模式：本地模式下，Spark 作业是在单机使用非并行模式执行。

（2）单机（Standalone）模式：Standalone 模式运行的 Spark 集群对不同的应用程序采用先进先出（FIFO）的顺序进行调度。默认情况下，每个应用程序会独占所有可用节点的资源。

（3）伪分布式（local-cluster）模式：伪分布式模式在单机环境下模拟了 Standalone 模式。在伪分布式模式下资源的调度流程与 Standalone 模式完全相同，只是资源调度的不是实际的物理节点，而是在单机上运行的伪分布 Spark 集群。

（4）YARN 模式：Spark 集群运行在 YARN 资源管理框架上。在 YARN 模式下，可以为特定的应用程序分配一定数量的 Executor，同时还可以对每个 Executor 使用的内存大小和占用的 CPU 内核数据进行设定。

（5）Mesos 模式：Spark 集群运行在 Mesos 资源管理框架上。在 Mesos 模式下，既可以配置 Spark 集群使用静态资源的分配策略，又可以配置集群动态共享 CPU 内核的分配策略。在动态共享 CPU 内核分配策略下，每个独立的应用程序还会分配固定的内存资源，但当该应用程序所占用的某个机器的 CPU 内核处于空闲时，其他应用程序可以使用该空闲的 CPU 内核资源。

3.2.2　Spark 作业执行过程

在了解 Spark 集群中各个节点功能后，接下来让我们来了解下一个 Spark 程序提交到 Spark 集群后，这些节点是如何协调工作完成 Spark 程序执行的。Spark 程序从提交到集群到执行的过程如图 3-3 所示。

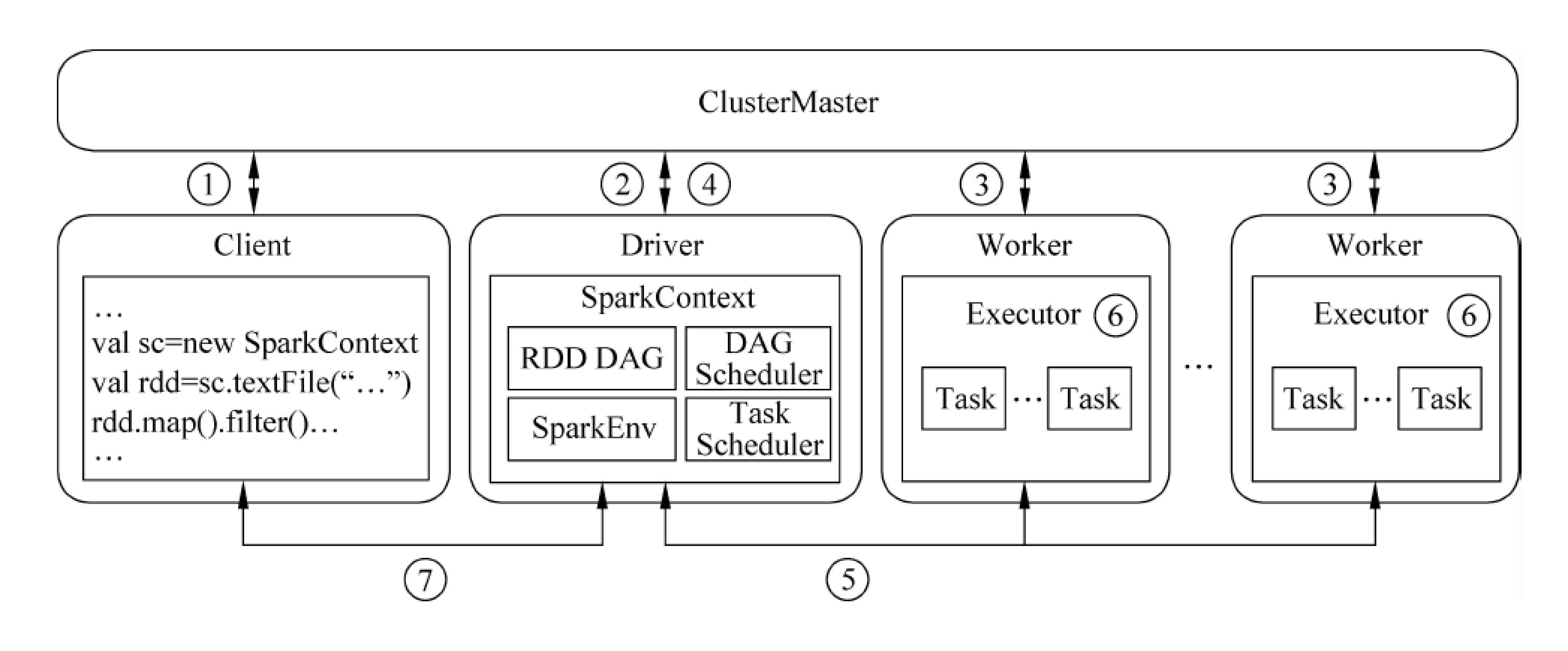


图 3-3　Spark 作业执行过程

（1）首先，用户会根据自己实际的需求编写一个 Spark 程序，就像我们之前编写的用于统计文件中单词个数的 WordCount 程序一样，在应用程序中包含了一个用户定义的主函数，在主函数内部实现 RDD 创建、RDD 转换、RDD 存储等操作，以完成用户的实际需求。之后用户会将完整的 Spark 程序提交到集群，申请 Spark 集群资源并行执行该程序。集群收到用户提交的 Spark 程序后，对应的 Driver 进程将被启动，负责响应执行用户定义的主函数。Driver 进程的启动可以有两种方式：①Driver 进程在客户端启动；②Master 节点指定一个 Worker 节点启动 Driver 进程，来充当 Driver 节点的角色。

（2）Driver 进程响应执行用户定义的主函数后，会发现主函数内部包含了 RDD 创建、RDD 转化、RDD 存储等操作，而这些操作都需要整个集群并行完成，于是 Driver 进程就会与 Master 节点进行通信，通过 Master 节点（它管理着集群上可以用于并行执行任务的全部资源）申请执行程序所需的资源。

（3）在 Spark 集群中的所有 Worker 节点都会向 Master 节点注册自己的计算资源，以便 Master 合理调配计算资源供应用程序使用，并且 Master 节点会通过心跳检测来检查已注册的 Worker 节点是否存活。在收到 Driver 进程的资源请求后，Master 节点会命令已注册的 Worker 节点启动 Executor 进程。

（4）如果 Worker 节点是正常存活的，并且该 Worker 节点上 Executor 进程启动成功，那么，Master 节点就会将这些启动的资源通知到 Driver 进程，使 Driver 进程使用集群中的那些资源来并行完成该 Spark 程序。

（5）Driver 节点根据 RDD 在程序中的转换和执行情况对程序进行分割，分割过程会遵循 Spark 的一些内部机制，具体的分割规则我们会在后面详细介绍。Driver 节点将分割后的任务发送给已经申请的多个 Executor 资源，每个 Executor 进程负责独立完成分给它的那部分计算任务，并将执行的结果反馈给 Driver 节点。Driver 节点负责了解每一个 Executor 进程的完成情况，以统一掌控整个 Spark 程序的完成情况。

（6）Worker 节点上运行的 Executor 进程是作业的真正执行者，在每个 Worker 节点上可以启动多个 Executor，每个 Executor 单独运行在一个 JVM 进程中，每个计算任务则是运行在 Executor 中的一个线程。最后，Executor 负责将计算结果保存到磁盘中。

（7）Driver 通知 Client 应用程序执行完成。

在上面的过程中，最重要的是 3 个步骤，如图 3-4 所示。

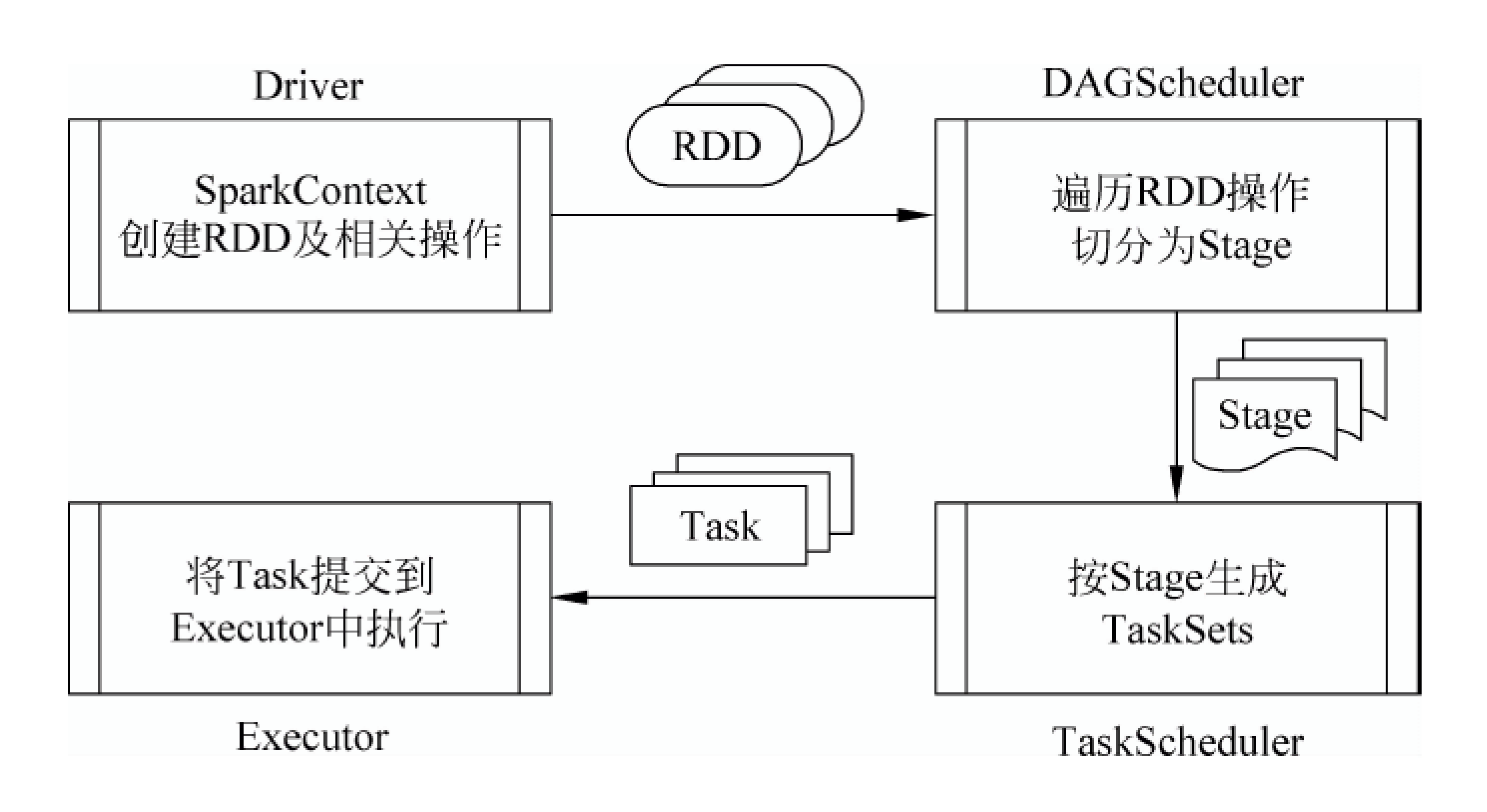


图 3-4　Spark 作业执行的 3 个关键步骤

（1）第一个步骤是生成 RDD 的过程。Driver 节点根据 Spark 程序中的代码逻辑创建出 RDD。RDD 被创建之后，就可以进行一系列的算子操作了。Spark 本身对 RDD 的操作模式是惰性计算。在惰性计算机制中，尽管每一次算子操作会将 RDD 转换为另一个新的 RDD，并且逻辑上会顺序的执行这一系列运算，但这些 RDD 的操作并不是

立即执行的,而是会等到出现行动算子(Action Operation)时才触发整个 RDD 操作序列,将之前的所有算子操作形成一个有向无环图(Directed Acyclic Graph,DAG),每个有向无环图再触发 Spark 执行一个作业(Job)。例如,在我们前面的 WordCount 示例程序中,只有到程序最后一行执行 saveAsTextFile 行动算子时,Spark 才对 RDD 进行真正的处理,将之前的 flatMap、map、reduceByKey 和 saveAsTextFile 这些算子操作连成一个有向无环图,并向 Spark 提交该作业。采用惰性计算的优势在于,相关的操作序列可以进行连续计算,而不用为存储的中间结果离散地独立分配内存存储空间。这样可以节省存储空间,同时,也为之后对 RDD 变换操作的优化提供了条件。

(2) 第二个重要过程是生成 Stage。Driver 节点中的 DAGScheduler 实例会对有向无环图中节点间的依赖关系进行遍历,将所有操作切分为多个调度阶段(Stage)。

(3) 第三个重要过程是生成 Task。每个 Stage 还需要转换为任务(Task)在集群中的 Worker 节点执行,因此,还要由 Driver 节点中的 TaskScheduler 实例将 Stage 转换为 Task,并提交到 Worker 节点的 Executor 进程中执行。

在下面的小节中,我们分别对以上 Spark 程序执行的具体步骤进行详细说明。

3.2.3 应用初始化

将每一个应用程序提交到 Spark 环境中运行时,都需要先完成一个应用初始化过程,其主要工作是进行配置加载和作业初始化,最终创建出 SparkContext 实例,以支持程序的链接集群、创建 RDD、变化 RDD 等后续工作。触发 Spark 应用初始化有两种情况。

(1) 使用 Spark-shell 执行 Spark 程序,在 Spark-shell 交互式环境启动时,就会自动为用户完成 Spark 的配置工作,并自动创建 SparkContext 来连接 Spark 集群,在我们看到 Spark-shell 的命令行输入窗口,即已完成应用初始化过程。此时,我们可以“sc”对象执行 Spark 操作,这里的“sc”对象就是 SparkContext 的首字母缩写。

(2) 使用 spark-submit 提交 Spark 程序的方式。我们可以通过 Spark 提供的 spark-submit 脚本将应用程序提交给 Spark 集群处理。用户可以首先将编写好的应用程序进行打包生成 Jar 文件,然后可以通过配置好的 spark-submit 脚本将应用程序提交给 Spark 集群处理。在 spark-submit 脚本中,用户可以根据自己的需要和集群的实际情况配置多个参数。下面就是运行在 YARN 集群上的一个 spark-submit 脚本的示例。

spark - submit 脚本示例

```
./bin/spark - submit - - master yarn - cluster - - class wordcount - -
deploy - mode client \
 - - name wordcount - - executor - memory 1g - - executor - cores 1 \
… \                          #其他参数
/home/wordcount.jar \        #程序 jar 包地址
/user/Input/Readme.txt \#程序参数
/user/Output \               #程序参数
```

上面脚本中的主要参数如下。

√ master：指明集群中的 Master 节点，之前我们已经介绍过，Spark 集群可以运行在多种模式下，因此，在这里需要根据不同的模式设置不同的 Master 节点。

√ class：用户的应用程序大多是使用 Scala 或 Java 语言编写的，在一个应用程序的 Jar 包中，可能就会包含多个类文件，class 指明了应用程序的入口类。

√ deploy-mode：指明 Driver 节点的运行地点，既可以选择在用户的客户端电脑运行 Driver 节点，也可以将 Driver 节点运行在集群中的一个节点上。

√ name：指定应用程序的名称，该名称会出现在 Spark 的 Web UI 中，方便用户找到自己定义的应用程序。

√ executor-memory：指明需要配置 Executor 的内存大小，Spark 对于 RDD 的操作都是基于内存的，因此，Executor 的内存大小设置直接影响到程序的性能。

√ executor-cores：用户可以通过设定来指定 Executor 使用的服务器 CPU 内核数目，在 YARN 模式下，可以通过 executor-cores 加数量参数指定每个 Executor 使用的内核数，而在 Mesos 模式下需要使用 total-executor-cores 加数量参数来为所有 Executor 指定可以使用的内核总数。

√ 程序地址：指明了应用程序的 Jar 包地址，地址既可以是集群上的 HDFS 地址，也可以是本地的文件地址。

√ 程序参数：指明了传给主函数类的参数。本例中包含两个参数，程序的输入地址和输出地址。

在 spark-submit 脚本可以设置的参数还有很多，通过在 spark-submit 脚本设置不同的参数可以使应用程序运行的更加高效，这里我们就不一一列出了。

通过 spark-submit 脚本，用户将 Spark 程序提交给了 Spark 集群，Spark 集群会根据

spark-submit 脚本中的 deploy-mode 设定选取一台主机运行 Driver 程序。Driver 程序根据 spark-submit 脚本中 class 设定的程序入口类进入应用程序。每个应用程序入口类的主函数内都会包含一个 SparkContext 实例。SparkContext 是整个应用程序连接集群的接口,它将告诉应用程序如何访问一个 Spark 集群。SparkContext 主要负责的工作有以下几方面。

（1）接受 SparkConf 参数：在 SparkContext 初始化时,Spark 运行环境会将 SparkConf 相关的配置参数传递给 SparkContex,用于配置应用运行时的属性,例如：要连接的 Master 节点、应用程序的名称、sparkHome、环境变量等。需要注意的是,SparkConf 和 spark-submit 脚本中的参数都可以对应用运行时的参数进行设置。但是,SparkConf 的优先级要高于 spark-submit 脚本,如果你用 SparkConf 对象设置了集群参数,则将覆盖掉 spark-submit 脚本内的设置。

（2）创建 SparkEnv 运行环境：Spark 的运行离不开一些重要的管理模块,像 BlockManager、CacheManager 等,SparkEnv 用于根据之前设置的集群参数创建这些管理模块。

（3）资源申请：整个应用程序通过 SparkContext 与集群连接并向 Spark 集群资源管理器 cluster manager 申请运行 Executor 资源,一旦资源申请成功,每个应用程序就会获得分布在不同节点上的 Executor 资源,SparkContext 将应用代码发送给各个 Executor,由 Executor 实际完成应用程序的计算任务。

（4）创建 SparkUI：Spark 为每一应用程序都提供一个单独的 web UI 管理界面。

（5）创建 TaskScheduler：TaskSchedular 的初始化会根据 Spark 运行的模式不同而不同。初始化启动后 TaskScheduler 负责每个任务的实际物理调度。

（6）创建 DAGScheduler：DAGScheduler 根据创建的 TaskScheduler 进行创建,DAGScheduler 负责接受提交的计算任务,同时负责任务的逻辑调度。

（7）提供函数方法：SparkContext 还提供了多个重要的函数方法以操作数据,例如,在我们前面的示例中用到的 textFile 方法,用于从 HDFS 路径读取数据文件,转化为 RDD。

一旦 SparkContext 创建成功,即完成了 Spark 应用的初始化,此时,就可以通过访问 Driver 节点的 4040 端口查看该应用程序的状态。如图 3-5 所示,我们可以看到该程序启动了一个 Driver 程序,并且成功申请了 3 个 Executor 资源。

Spark 1.4.0 Jobs Stages Storage Environment Executors Spark shell application UI

Executors (4)

Memory: 0.0 B Used (1061.6 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	10.106.129.44:51358	0	0.0 B / 265.4 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
1	10.106.129.44:47209	0	0.0 B / 265.4 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	10.106.129.44:46317	0	0.0 B / 265.4 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
driver	10.106.129.44:36556	0	0.0 B / 265.4 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

图 3-5 Spark driver 启动

3.2.4 构建 RDD 有向无环图

Spark 应用初始化并通过 SparkContext 函数读取输入数据生成第一个 RDD 后，后续的 Spark 程序就是通过 RDD 算子对 RDD 进行一次又一次的变换，最终得到计算结果。因此，一个 Spark 应用可以看作一个由“RDD 创建”到“一系列 RDD 转化操作”再到“RDD 存储”的过程。在这个过程中，每个 RDD 自身是不可变的，程序是通过将一个 RDD 转化为另一个新的 RDD，经过一个像管道式的流水线一样一级一级的变换，将初始 RDD 生成新的中间 RDD，最终生成你想要的 RDD 并输出。为了完成这个转换过程，Spark 首先要将我们编写的程序或脚本中的 RDD 操作语句构建出一个 RDD 有向无环图（Directed Acyclic Graph，DAG），以进行后续的拆分和调度。

有向无环图是一种非常重要的图论数据结构。如果一个有向图无法从任意顶点出发经过若干条边回到该点，则这个图就是一个有向无环图。有向无环图的这种连通关系常被用来表达节点间的时序关系。其中，节点表示某种任务，而边表示任务间的约束转化，任务间的转化会形成一个有序无环的任务序列，后续的任务会依赖前面任务的执行，通常后续的任务被称为子任务，而其依赖的任务称为该任务的父任务。对应到 Spark 的 RDD 的有向无环图中，顶点代表了 RDD 及产生该 RDD 的操作算子，有方向的边代表算子间的转化。

RDD 有向无环图中，新产生的 RDD（称为子 RDD）都是通过若干个 RDD（称为父 RDD）转换产生的，因此，子 RDD 的内容是依赖于父 RDD 的内容的。Spark 给 RDD 间

的这种依赖关系起了个名字叫作 Lineage,其英文含义为可以通过族谱从一个人追溯到之前的几代人,我们可以称之为“世系”。因为 Spark 保留了每个 RDD 的世系信息,所以可以从某个 RDD 查找到其父 RDD,并进而找到最原始的那个 RDD。为了确保世系关系的唯一性,Spark 规定了在 RDD 间不存在直接或间接的循环依赖关系,这也就是 Spark 的 RDD 转化关系构成的图是有向无环图的原因。

那么,Spark 中为什么要保留世系信息,并支持通过一个 RDD 逐级往上查找 RDD 呢? 我们知道,Spark 是一个分布式并行计算系统,这就不可避免地存在某个节点宕机、数据网络传输丢失等问题。为了在发生这些问题时,仍能保证整个 Spark 应用能完整执行完成,就必须有一种容错机制。世系信息的记录正是为了达到这个容错目的。当某个 RDD 的部分数据丢失时,Spark 会根据记录的世系关系找到该 RDD 的父 RDD 以及更上级的 RDD,只需要将该 RDD 依赖的上级 RDD 重新计算就可以将该 RDD 进行恢复。

因此,Spark 的 RDD 有向无环图构建过程,就是不停地将 Spark 代码中一系列的 RDD 转化操作以世系关系的形式记录下来。随着代码的逐行扫描,世系信息不断增长。但是,在这个过程中,Spark 还并不会真正执行这些转换,而仅仅是进行记录,直到出现一类称为行动(Action)算子的特殊操作,才会触发作出实际的 RDD 操作序列的动作,将行动算子之前的所有算子操作形成一个有向无环图的作业(Job),并将该作业提交给集群申请进行并行作业处理。这种延迟处理的方式被称为惰性计算(Lazy Evaluation)。采用惰性计算的好处在于,相关的操作序列可以进行连续计算,而不用担心存储的中间结果需要独立分配内存存储空间,这样节省了存储空间。同时,大型的、复杂的运算作业被延迟到真正需要计算时才被计算,减少了大型作业占用内存的时间。

经过前面的这些过程,一个 Spark 程序就被分解为了多个作业提交给 Spark 集群。为了更加直观地展示这一过程,我们仍以已经熟悉的 WordCount 程序为例进行说明。在 WordCount 程序代码中,只有在运行到最后一行 saveAsTextFile 行动算子时,Spark 才真正对 RDD 转换过程进行处理。Spark 运行环境会将之前的 flatMap、map、reduceByKey 和 saveAsTextFile 算子操作连成一个有向无环图,并向 Spark 提交该作业。由于整个程序只有一个行动算子,因此,程序只会提交一个作业。我们可以在 Spark 的 Web UI 中看到生成的 RDD 有向无环图,如图 3-6 所示。图中每一个蓝色的矩形块代表了一个 RDD,而蓝色矩形块上的文字代表了产生该 RDD 的算子操作。

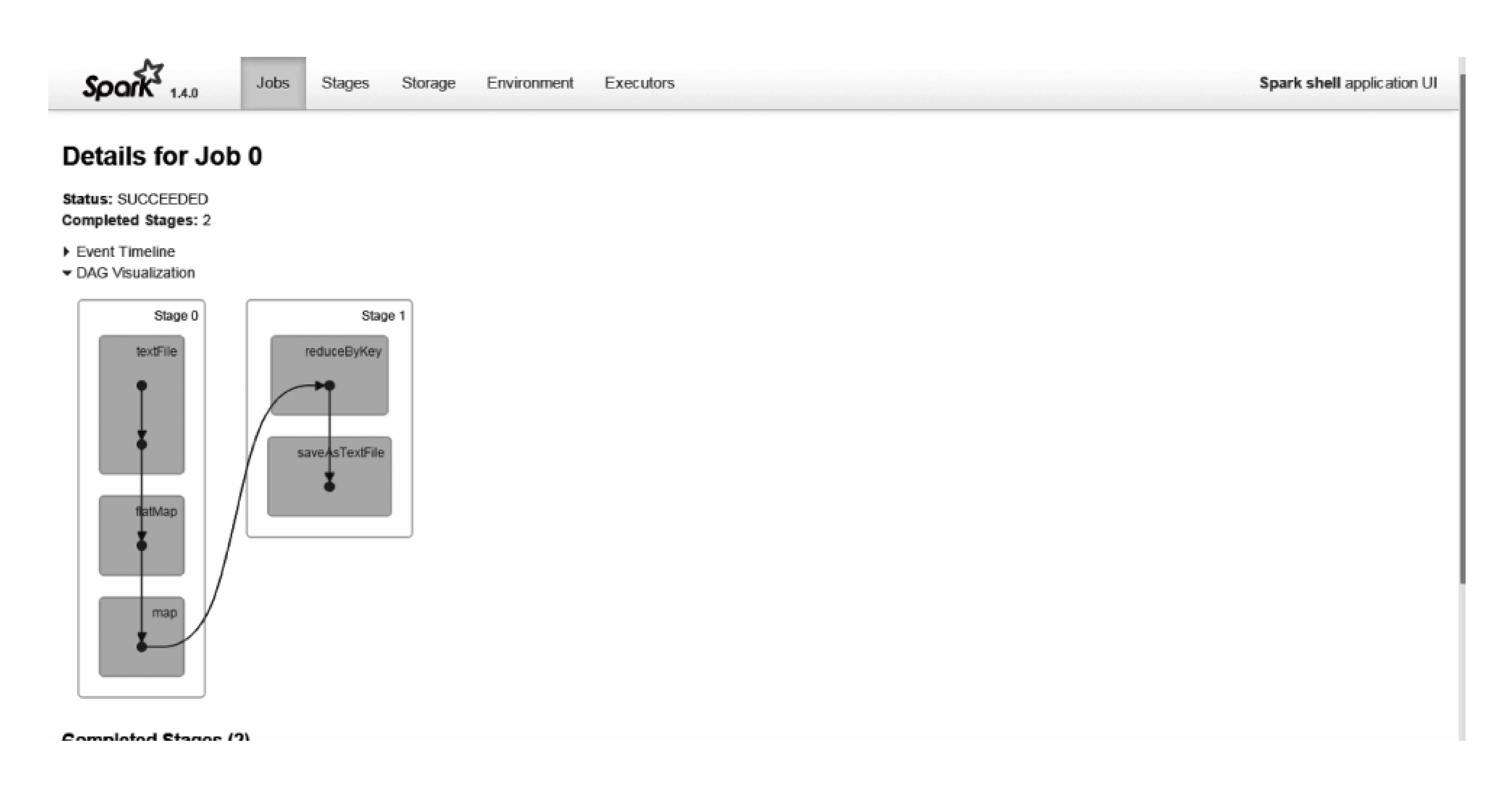


图 3-6 RDD 有向无环图

3.2.5 RDD 有向无环图拆分

在前面我们看到，一个 Spark 应用程序会被分解为多个作业（Job）提交给 Spark 集群进行处理。然而，作业并不是应用程序被拆分的最小计算单元，Spark 集群在收到作业后还要将作业继续进行两次切分和规划，并进行相应的调度。第一步是将作业按照 RDD 转换操作切分为最小处理单元，即任务（Task），第二步是对任务进行规划，生成包含多个任务的阶段（Stage）。这两步工作都是由 SparkContext 创建的 DAGScheduler 实例进行，其输入是一个作业中的 RDD 有向无环图，输出为一系列任务，因此，我们将这一步骤称为 RDD 有向无环图拆分。

在前面我们已经了解到，在 RDD DAG 中，子 RDD 与父 RDD 间是存在依赖关系的。与 Hadoop 类似，Spark 也是一个分布式计算环境，因此，在 Spark 中进行计算的每一个数据单元 RDD 也是可以被切分为更小的数据块在不同的计算节点中处理的，这样的数据块就是分区（Partition）。RDD 在进行转换的过程中，就是以分区为最小单位进行处理的。因此，由于存在依赖关系的父子 RDD 间进行转换的分区对应关系不同，RDD 间的依赖也分为两个类型：窄依赖和宽依赖。图 3-7（a）、图 3-7（b）中分别展示了这两种依赖。图中的每个矩形为一个 RDD，而椭圆块则为一个分区。这两种依赖的区别如下。

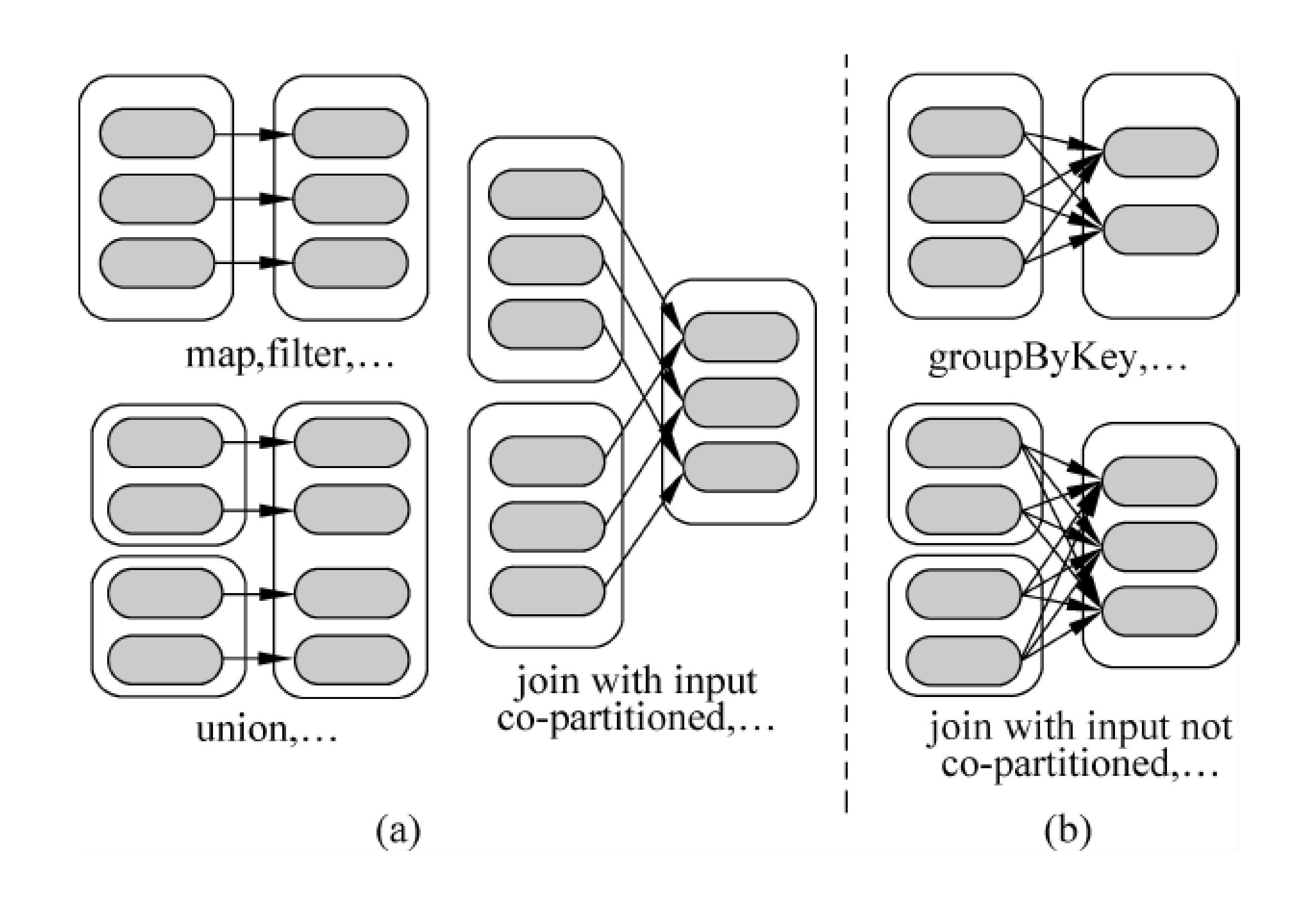


图 3-7 窄依赖与宽依赖

√ 窄依赖：父 RDD 的一个分区，在 RDD 转换过程中，最多只会被一个子 RDD 的分区使用，例如图（a）中的 map 转换、union 转换，和先进行 co-Partition 操作的 join 转换。

√ 宽依赖：父 RDD 的一个分区，在 RDD 转换过程中，会被多个子 RDD 的分区使用，例如图（b）中的 groupByKey 转换和没有做 co-Partition 操作的 join 转换。

从定义和图中可以看到，这两种依赖关系存在本质上的区别。

√ 窄依赖的分区间的逻辑关系非常明确，对于分区间是一对一的转换（例如 map、union），可以在一个计算节点内进行，并且如果有多个这样的窄依赖转换，可以在一个节点内流水线般执行。对于分区间有多对一（多个父 RDD 分区对一个子 RDD 分区）的窄依赖转换（例如先进行 co-Partition 操作的 join 转换），也可以在多个节点间并行执行，相互间没有影响。同时，在窄依赖的子 RDD 计算过程中某个分区出错时，只需要重新取得或计算父 RDD 对应的分区，即可恢复子 RDD 的这个分区。

√ 与此相反，宽依赖则要依赖父 RDD 的所有分区数据，才能计算得到子 RDD，因此，需要进行类似 Hadoop 中 MapReduce 的数据 Shuffle 过程。这就必然会带来网络开销、中间结果存储等一系列开销较大的问题。同时，在计算过程中出错时，也必须取得和计算全部父 RDD 的数据才能恢复，其代价远大于窄依赖。

由于窄依赖和宽依赖在转换和容错时存在巨大的差异，因此，Spark 将应用程序拆分为任务后分配到集群运行时，并不是直接将一个一个的转换操作对应的任务直接进行分配，而是加入了一个对任务进行规划的过程，以将适合放在一起执行的任务合并到一个阶段中。这一过程由 DAGScheduler 实例完成，其原则是如果子 RDD 到父 RDD

是窄依赖关系,就可以对其操作进行优化,首先将多个算子操作一起处理,最后再进行一次统一的同步操作,这样既减少了大量的全局同步,又无须存储很多中间结果。而对应宽依赖,则尽量切分到不同的阶段中,以避免过大的网络传输和计算开销。为了达到这一目标,应用程序向 Spark 提交 Job 作业后,DAGScheduler 会遍历 RDD 有向无环图。在遍历过程中,对于遇到的连续窄依赖 RDD 转换,则尽量多地放入同一个阶段中。一旦遇到一个宽依赖类型的 RDD 转换操作,则生成一个新的阶段。这一过程一次进行,直到遍历完整个 RDD 有向无环图。图 3-8 展示了一个示例过程。由 A 转换为 B 是一个宽依赖类型的 groupBy 操作,因此,生成了 Stage 1。而由 C 转换为 D 的 map 操作,由 D、E 转换为 F 的 union 操作,都是窄依赖类型,因此,被放入同一个 Stage 2。在遇到由 B、F 转换为 G 的宽依赖 join 操作时,则生成一个新的 Stage 3。

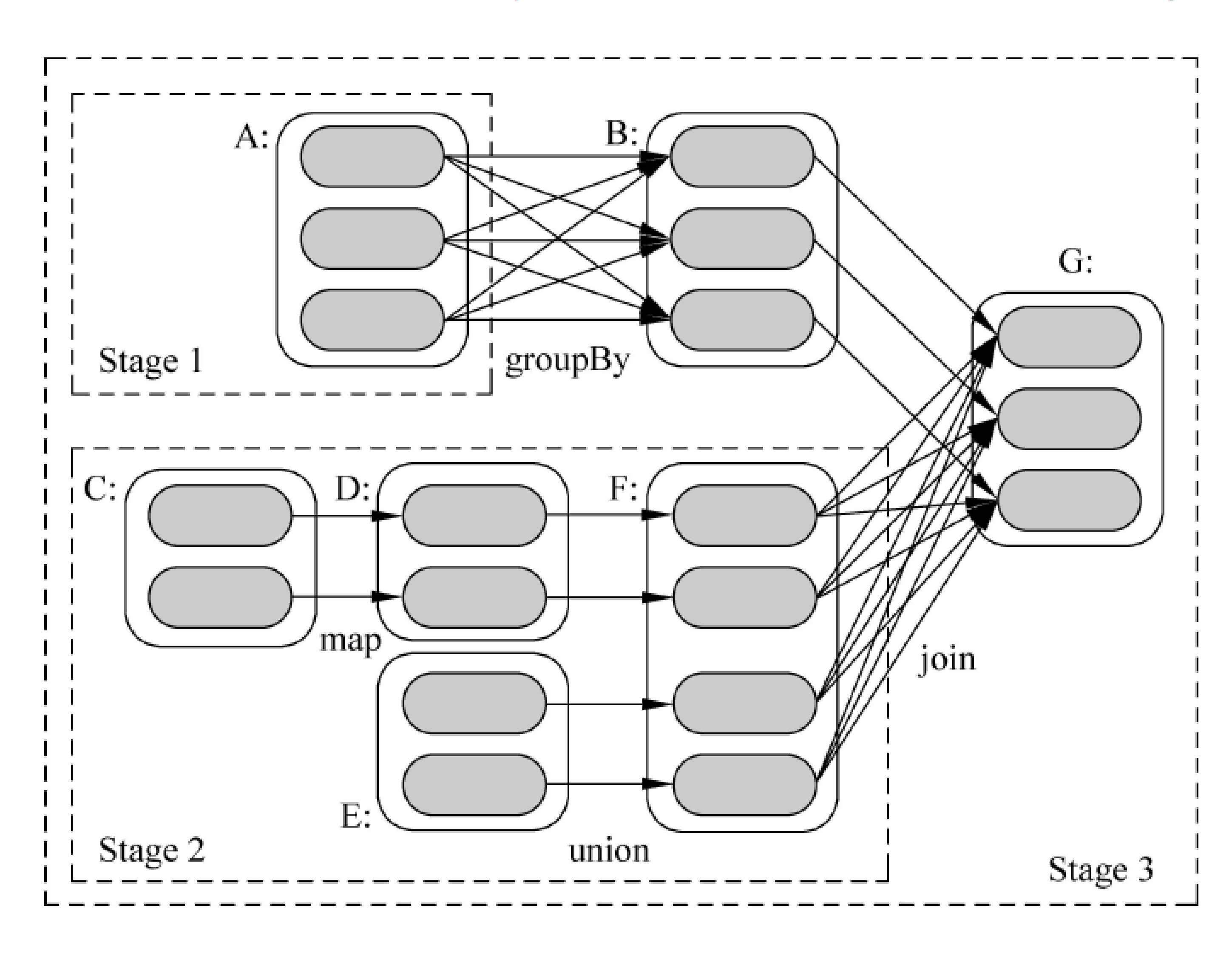


图 3-8 Stage 划分

通过这样的一个过程,DAGScheduler 实现了将依赖链进行分割的操作。整个依赖链被划分为多个 Stage 阶段,每个 Stage 内都是一组相互关联、但彼此之间没有 Shuffle 依赖关系的任务集合,称为任务集(TaskSet)。每个 TaskSet 包含多个任务。DAGScheduler 会根据分区的个数,来具体确定会生成多少个任务。一个分区对应一个任务,每个 Task 会在对应的数据分区上进行一系列的数据处理,同一阶段任务的执行是并行执行的。作业(Job)、阶段(Stage)、任务集(TaskSet)和任务(Task)之间的关系如图 3-9 所示。

DAGScheduler 不仅负责将作业分割为多个阶段,还负责管理调度阶段的提交。DAGScheduler 维护了 3 个集合用以存储阶段的执行状态。

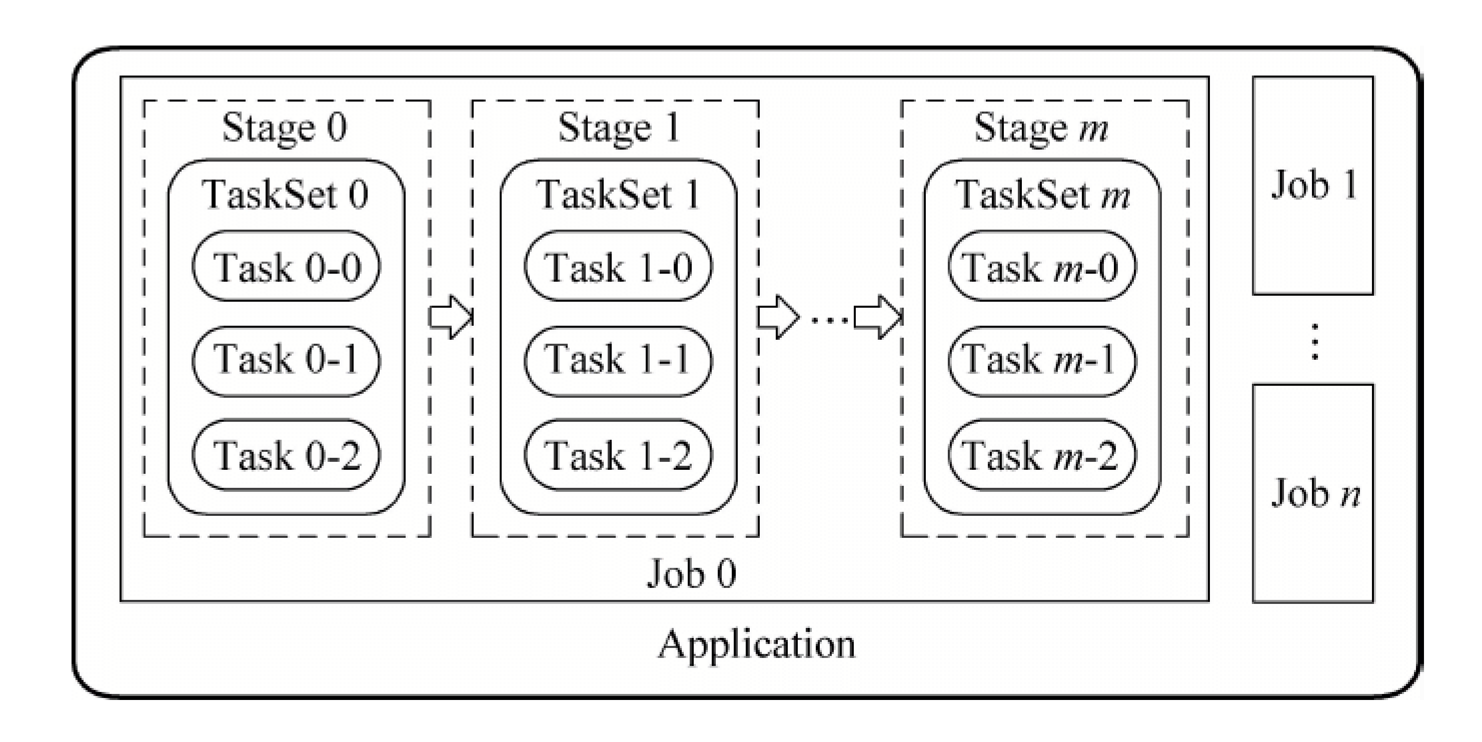


图 3-9　Job、Stage、TaskSet、Task 关系图

（1）waitingStages 集合：同 RDD 的父子依赖关系一样，在一串相互依赖的 Stage 中，后续的 Stage 被称为子 Stage，而其依赖的 Stage 称为该 Stage 的父 Stage。如果一个 Stage 的父 Stage 尚未完成，则 waitingStages 集合将负责记录该子 Stage。

（2）runingStage 集合：为了防止 Stage 的重复提交执行，runingStage 集合中保存正在执行的 Stage。

（3）failedStage 集合：failedStage 集合中保存了执行失败的 Stage，这些失败的 Stage 需要重新提交执行。

DAGScheduler 会根据 Stage 的运行状态合理调度所有 Stage 提交到集群。DAGScheduler 为每一个 Stage 分配一个 StageId，用以表示 Stage 的优先级，StageId 越小优先级越高，越应该最先提交给集群执行。由于 DAGScheduler 是反向遍历整个 RDD 依赖链的，因此，由最后一个 RDD 生成的 FinalStage 的 StageID 最小，应该首先被提交。但是，DAGScheduler 对 Stage 的提交还需要参考 Stage 间的依赖关系以及 Stage 的执行情况，如果一个 Stage 的父 Stage 仍处于未完成的状态，则该 Stage 将被延迟提交直到等到该 Stage 的所有父 Stage 全部完成才被提交。因此，虽然 FinalStage 的 StageID 最小，但它首先需要等待其父 Stage 的提交。如果 FinalStage 的所有父 Stage 都已经提交并已产生可用的结果，则提交该 FinalStage。如果有任何一个父 Stage 的结果不可用，则尝试迭代提交当前不可用的父 Stage。该不可用的父 Stage 的提交遵循与 FinalStage 同样的判别标准。以此迭代，直到遇到可提交的 Stage，便将其提交给 Spark 集群首先执行。

为了更直观地理解 RDD 有向无环图拆分的过程，我们同样还是用已经熟悉的 WordCount 程序为例进行实例展示。在收到提交的作业后，Spark 会首先从 RDD 依赖链的最后一个 RDD 进行判断，当遍历到由 reduceByKey 产生的一个 ShuffleRDD 时，Spark 对整个依赖链进行了一次分割，由于在整个应用程序的依赖关系中只有一个宽依赖操作，因此，DAGScheduler 最终将 wordcount 程序切分成 2 个 stage。Web UI 中看

到生成的 Stage 如图 3-10 所示。

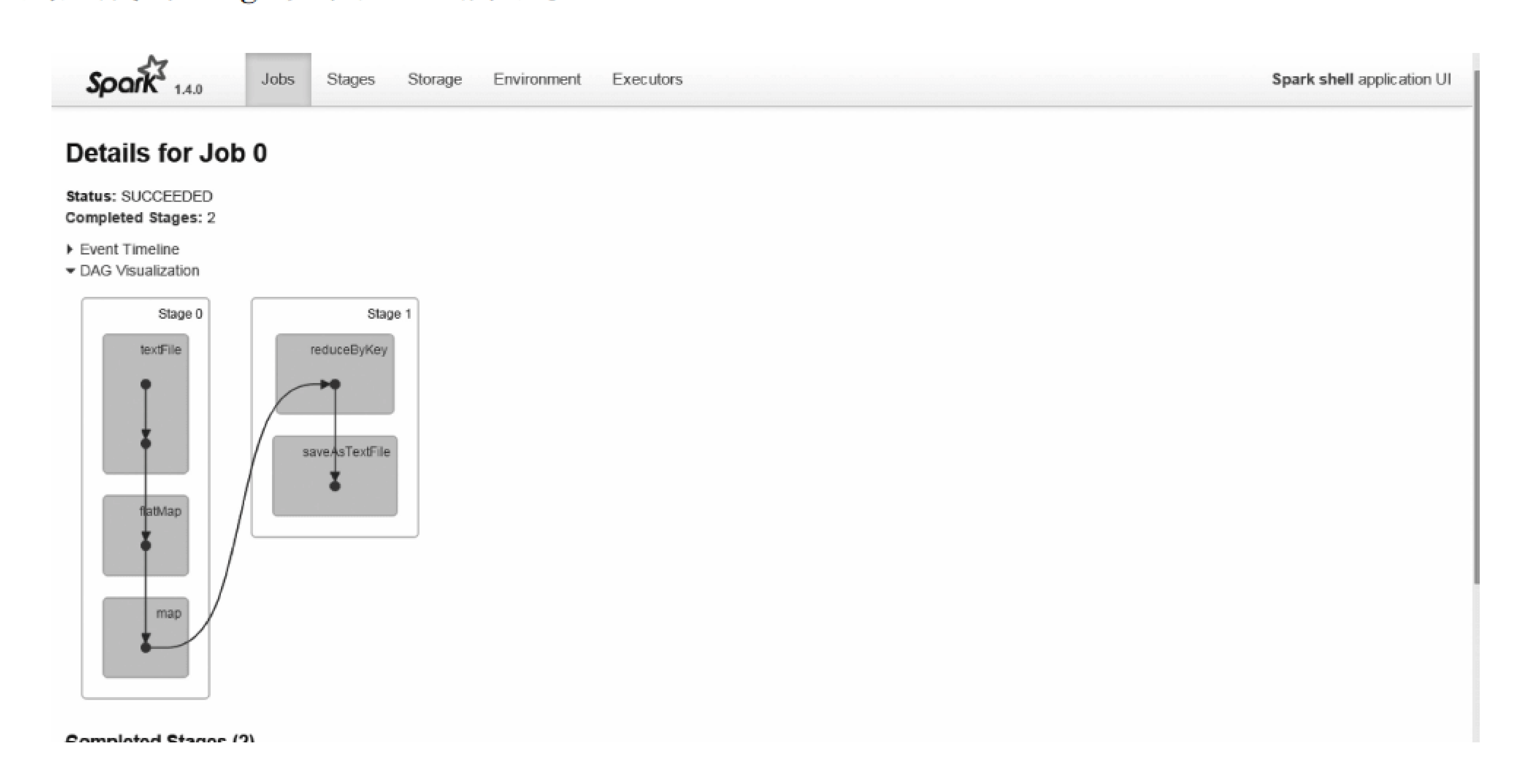


图 3-10　Stage 实例图

在 Spark 提供的时间轴的可视化视图中，我们可以看到一共有两个 Stage 的运行时间轴，虽然第二个 Stage 首先被 Spark 遍历，但是第一个 Stage 首先被执行，第二个 Stage 等待第一个 Stage 完成后才开始执行。这一过程如图 3-11 所示。

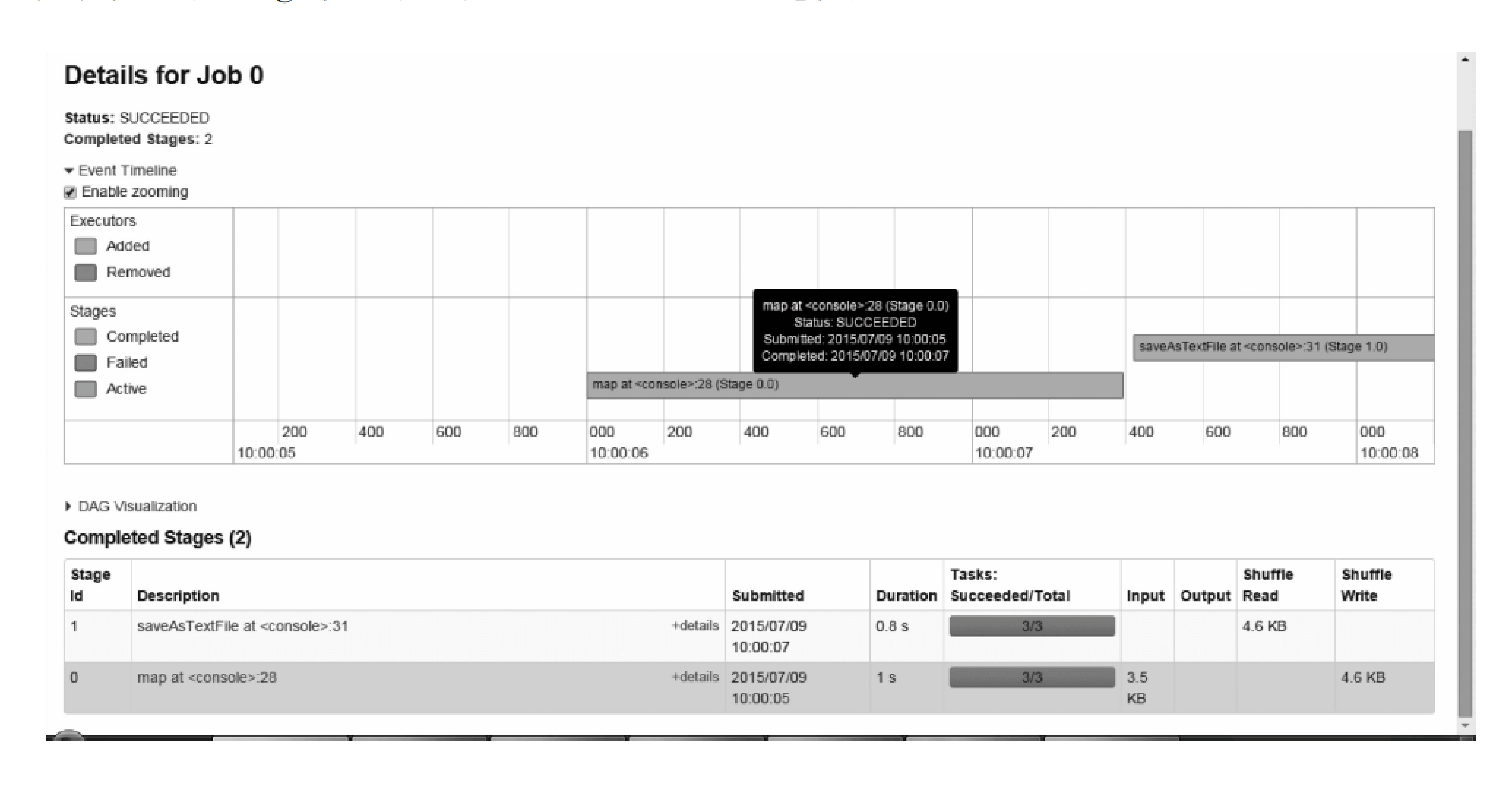


图 3-11　Stage 执行顺序

3.2.6　Task 调度

DAGScheduler 将建立好的 TaskSet 提交给 TaskScheduler 后，DAGScheduler 对任务的调度工作就算完成了。DAGScheduler 只是完成了将作业的有向无环图分割为

Stage，并生成了一个 Stage 执行的有序计算序列。而 TaskScheduler 真正决定了 Stage 中的每个 Task 由哪个物理节点执行。有向无环图、DAGScheduler 和 TaskScheduler 之间的关系如图 3-12 所示。

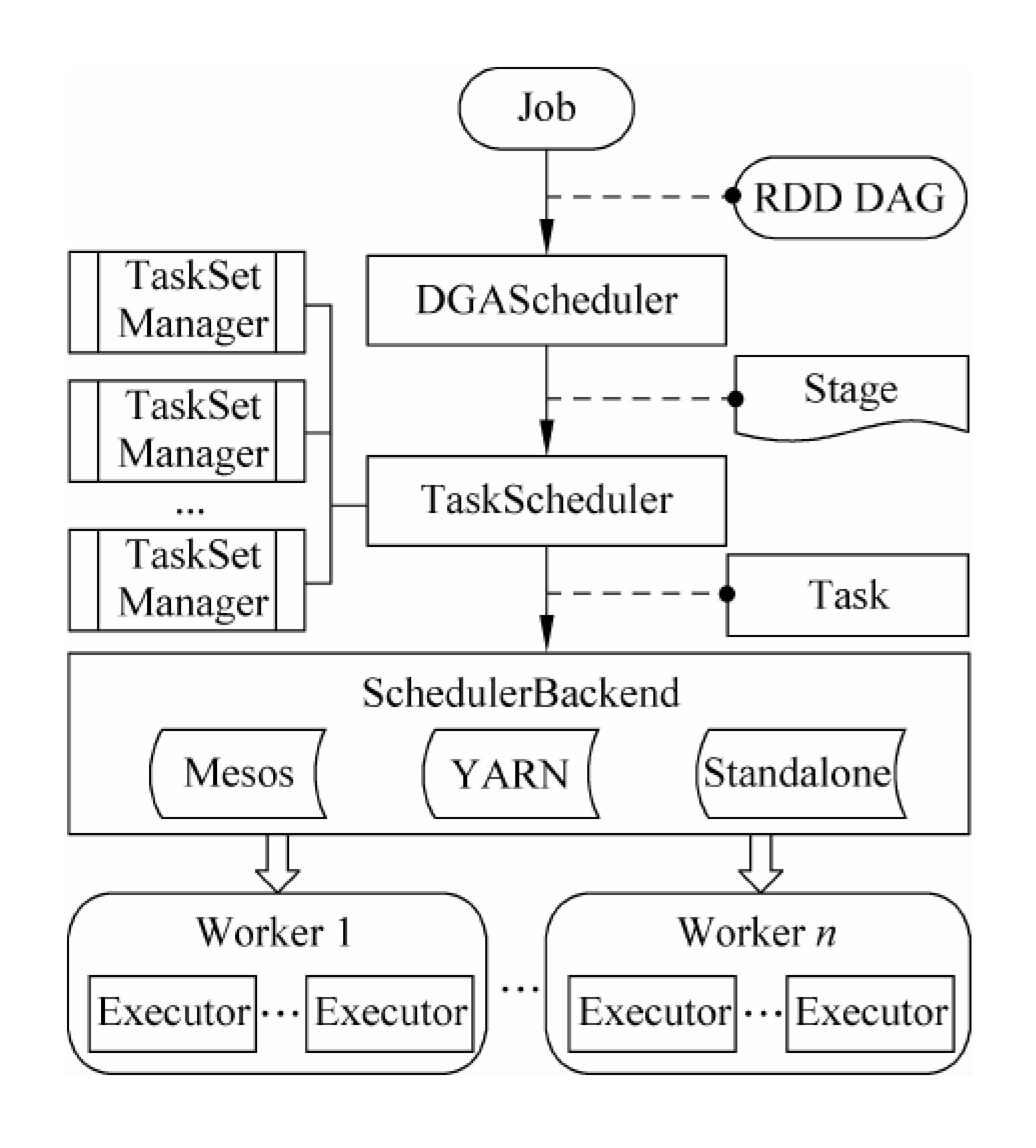


图 3-12 Task 调度

TaskScheduler 会为每一个收到的 TaskSet 创建一个 TaskSetManager。由 TaskSetManager 负责 TaskSet 中 Task 的管理调度工作，但是 TaskSetManager 并不直接与底层的实际物理节点通信，而是通过 TaskScheduler 这个中间层来协调每个 TaskSetManager 与底层物理节点的交互工作。一方面 TaskScheduler 会通过 SchedulerBackend 与底层的 Master、worker 节点进行通信，SchedulerBackend 是调度器的后台进程。TaskScheduler 会根据部署方式的不同而选择不同的 SchedulerBackend 来处理、了解可以分配给应用程序的物理资源的情况，SchedulerBackend 可以使用 Mesos、YARN、Standalone 等多种形式。另一方面，TaskScheduler 会将可用的物理资源提供给 TaskSetManager，由 TaskSetManager 确定每个 Task 应该在那个物理资源上执行，并将调度计划发给 TaskScheduler，由 TaskScheduler 将 Task 提交给 Spark 集群实际执行。TaskScheduler 将 Task 提交给 Spark 集群后还会跟踪 Task 的执行情况，如果 Task 提交失败，它会负责尝试重新提交该 Task。同一 TaskSet 中的多个 Task 是并行执行的，只有 TaskSet 中的所有 Task 全部完成该 TaskSet 才算真正完成，如果被分配到某个物理节点上的 Task 长时间未完成，拖延了整个 TaskSet 的执行时间，TaskScheduler 会选择一个新的可用物理节点执行该 Task。

TaskScheduler 为每个 TaskSet 启动的 TaskSetManager 负责管理每个 TaskSet 的内部调度任务，维护整个 TaskSet 的生命周期。TaskSetManager 的调度策略是基于位置感知的任务调度。TaskSetManager 会根据可用资源的情况尽量为每个 Task 选择最佳本地化（Locality）的 Executor 进行匹配。也就是说，Spark 会首先判断同一个 Stage 中首个 RDD 各个分区（Partition）的位置，每个 Task 是对一个分区的一串操作，Spark 会尽量将对一个分区的 Task 操作保持在同一地点。最理想的情况是将同一个 Task 的前后操作放到同一进程中，如果条件得不到满足可以退而寻求在同一计算节点中，如果还不能满足就会寻求在同一个机架运行的可能。最终 TaskSetManager 选择一个最为合适运行的计算资源，将该资源的申请交给 TaskScheduler。

一个应用程序最终会拆分为多个 TaskSet 任务集，由于有的 TaskSet 任务集间没有依赖关系，因此，同一时间可能会存在多个可运行的 TaskSet 任务集交给 TaskScheduler 进行调度。在这种情况下，Spark 可以采用两种调度模式：FIFO 和 FAIR。

- FIFO：FIFO 是先进先出的调度模式，是 Spark 默认的调度模式。FIFO 的调度模式如图 3-13（a）所示。FIFO 直接调度管理的是 TaskSet，每个 TaskSet 创建时都对应了一个 StageID 和 JobID，FIFO 调度根据 StageID 和 JobID 的大小来调度 TaskSet，相当于依据 StageID 和 JobID 的大小对 TaskSet 进行了两级排序，JobID 的大小为第一级排序，StageID 的大小为第二级排序，数值较小的将首先进入任务执行队列，优先被调度。这种先进先出的 FIFO 调度方式存在的一个缺点是当遇到一个耗时较长的大任务时，后续任务必须等待这个耗时任务执行完成才能得到可用的计算资源。
- FAIR：FAIR 是公平调度模式，调度模式如图 3-13（b）所示。在 FAIR 模式下每个计算任务具有相等的优先级，Spark 以轮询的方式为每个任务分配计算资源。FAIR 不像 FIFO 调度那样必须等待前面耗时任务完成后后续任务才能执行。在 FAIR 模式下，无论是短任务还是耗时任务、无论是先提交的任务还是后提交的任务都可以公平的获得资源执行，这样就提高了短任务的响应时间。同时，FAIR 调度模式比 FIFO 调度模式更加灵活，FAIR 调度模式为用户提供一个调度池的概念，用户可以将重要的计算任务放入一个调度池 Pool 中，通过设置该调度池的权重来使该调度池中的计算任务获得较高的优先级。除此之外，当有多个用户同时使用 Spark 集群时，FAIR 调度模式可以为每一个用户设置一个调度池，调度池间具有相同的优先级，这样就保证了每个用户可以平等

地使用 Saprk 计算资源。

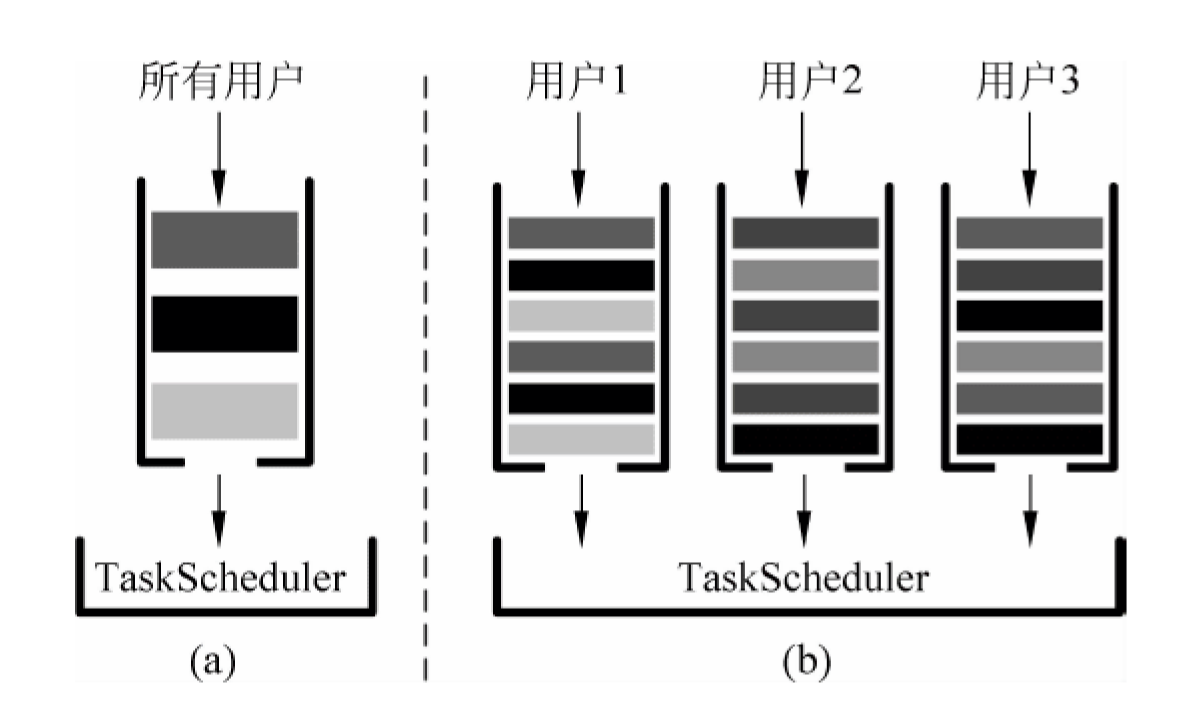


图 3-13 FIFO 与 FAIR 调度机制

3.2.7 Task 执行

通过上一小节的介绍我们了解到,TaskScheduler 会在集群中选择合适的计算资源提供给 TaskSetManager。TaskScheduler 在集群中选择的 Worker 节点需要满足两个条件:①Worker 节点内存大小必须可以满足应用程序设定的 executor-memoy 大小;②Worker 节点的内核数可以满足应用程序设定的 executor-core 数。如果 Worker 节点满足以上两个条件就会被 TaskScheduler 选中为可用的计算资源,在该 Worker 节点上就会启动 Executor 进程,等待 TaskScheduler 将合适的 Task 分发给它运行。

每一个满足条件的 Worker 节点会为每一个应用程序独立地启动一个 Executor 进程,由该 Executor 进程全权负责该应用程序的 Task 任务,集群中的多个 Worker 节点的 Executor 进程在收到 Task 后,就会开始相互独立的并行完成 Task 的计算任务。在每个 Worker 节点可以启动多个 Executor 进程来满足多个应用程序的请求,而每个 Executor 进程都单独的运行在 Worker 节点一个 JVM 进程中,每个 Task 则是运行在 Executor 中的一个线程。某个应用程序的 Executor 进程一旦被创建将一直运行,且它的资源可以一直被多批 Task 任务复用,资源不会在应用程序的计算过程中被释放,这避免了重复申请资源带来的时间开销值,资源会始终保留直到该 Spark 程序运行完成后才会释放退出。

与 DAGScheduler→TaskScheduler→Executor 的任务分发过程正好相反,Executor 会反向逐级向上报告任务的运行状态。如图 3-14 所示。

Executor 上对任务状态的标志主要有 4 种情况。

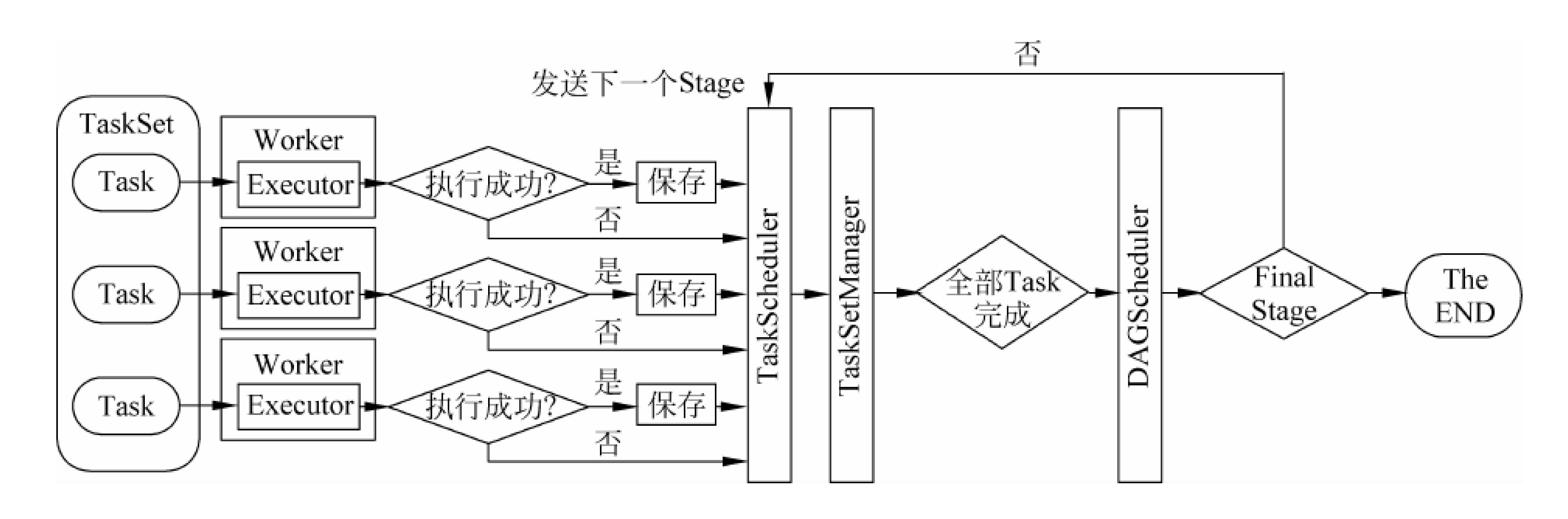


图 3-14　任务状态上报机制

（1）Executor 收到任务后，尚未完成对 Task 执行任务，此时 Task 的状态为 RUNNING。

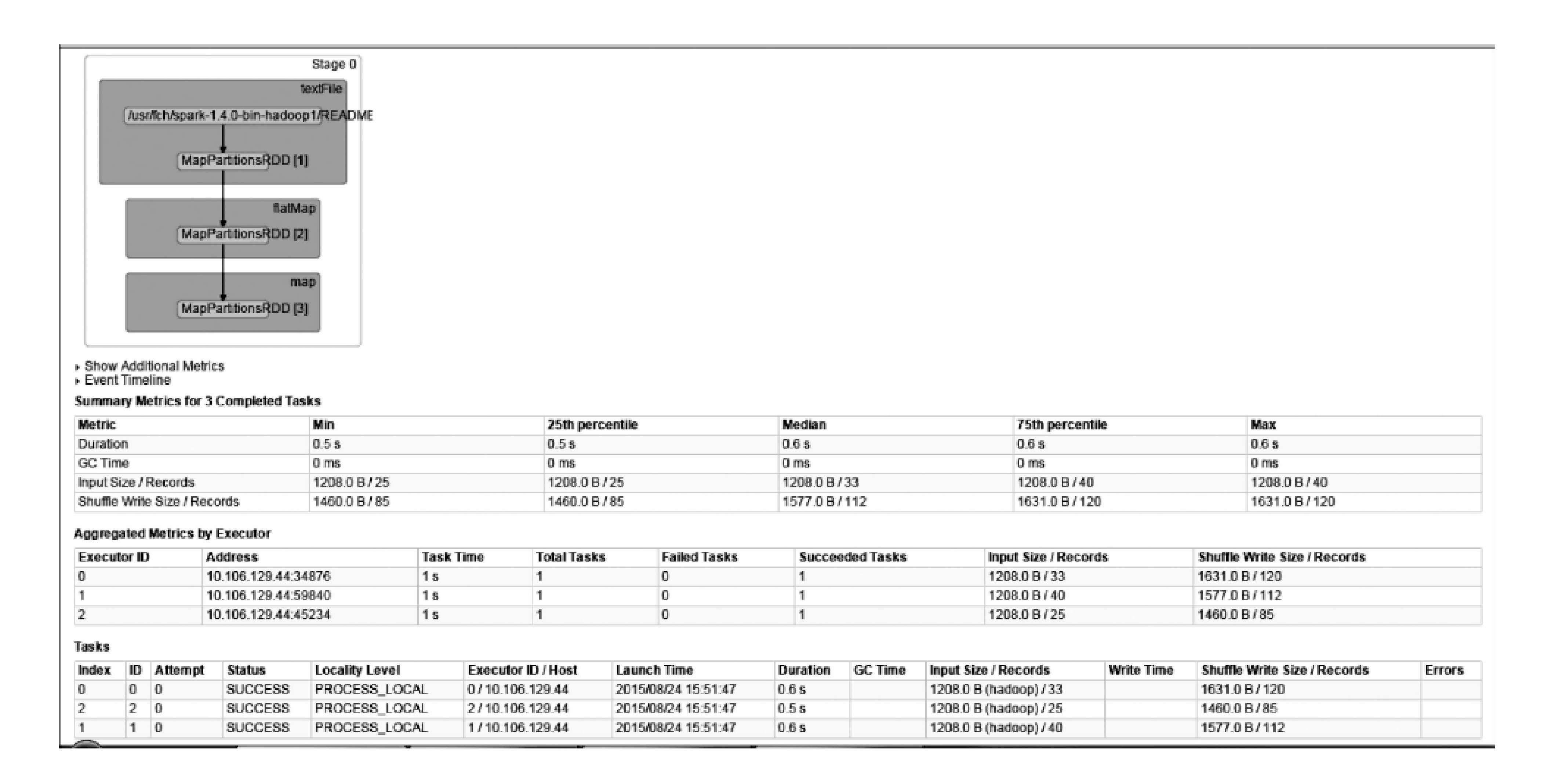
（2）如果 Task 运行过程中发生错误，此时，Task 的状态将被标记为 FAILED，Executor 会将错误的原因进行上报。

（3）如果 Task 在中途被人为或者其他原因 Kill 掉了，则此时 Task 的状态将被标记为 KILLED，Executor 会将 kill 的原因上报给 TaskScheduler。

（4）如果某个 Executor 进程的 Task 计算完成，此时 Task 的状态为 FINISHED，该 Executor 进程就会将结果进行保存，并将结果及状态信息反馈给 TaskScheduler，并等待 TaskScheduler 为其分配下一个 Task 任务。

TaskScheduler 收到 Executor 发来的结果及状态信息后，就会找到该 Task 对应的 TaskSetManager，将该 Task 的完成情况通知给对应的 TaskSetManager。如果 TaskSetManager 所管理维护的所有 Task 均已完成，TaskSetManager 会自动结束关闭，并将该 TaskSetManager 的运行结果告知 DAGScheduler。如果该 TaskSetManager 对应的 Stage 是 FinalStage，就将运输结果本身返还给 DAGScheduler。而如果对应的是中间的、由 Shuffle 操作而被分割开的 Stage，则返还给 DAGScheduler 的是运算结果在存储模块的相关位置信息，这些存储位置信息将作为下一个调度 Stage 的输入数据，下一个 Stage 读取该输入数据进行计算，就这样反复进行任务分发、任务执行和任务执行信息上报的过程直到全部任务完成，该应用程序也就执行完成了。

在 Spark 提供的可视化视图中，我们可以观察到全部 Task 的运行状态，以便我们了解任务的运行情况，图 3-15 是 wordcount 程序中第一个 Stage 的 Task 运行情况。由图中我们可以看出，第一个 Stage 的 3 个 Task 全部执行成功，没有 Failed Tasks。



Summary Metrics for 3 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.6 s	0.6 s	0.6 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Input Size / Records	1208.0 B / 25	1208.0 B / 25	1208.0 B / 33	1208.0 B / 40	1208.0 B / 40
Shuffle Write Size / Records	1460.0 B / 85	1460.0 B / 85	1577.0 B / 112	1631.0 B / 120	1631.0 B / 120

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
0	10.106.129.44:34876	1 s	1	0	1	1208.0 B / 33	1631.0 B / 120
1	10.106.129.44:59840	1 s	1	0	1	1208.0 B / 40	1577.0 B / 112
2	10.106.129.44:45234	1 s	1	0	1	1208.0 B / 25	1460.0 B / 85

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	0 / 10.106.129.44	2015/08/24 15:51:47	0.6 s		1208.0 B (hadoop) / 33		1631.0 B / 120	
2	2	0	SUCCESS	PROCESS_LOCAL	2 / 10.106.129.44	2015/08/24 15:51:47	0.5 s		1208.0 B (hadoop) / 25		1460.0 B / 85	
1	1	0	SUCCESS	PROCESS_LOCAL	1 / 10.106.129.44	2015/08/24 15:51:47	0.6 s		1208.0 B (hadoop) / 40		1577.0 B / 112	

图 3-15　WordCount 运行情况图

第4章

RDD 算子

在上一章中我们介绍了 Spark 的运行原理、机制和节点功能。我们可以看到，Spark 计算框架中的核心数据结构是 RDD，而要完成一个简单或复杂的计算，其关键就在于如何将数据生成为 RDD，然后对 RDD 进行变换和操作，以获得最终需要的计算结果。完成这一过程的关键就是 Spark 的另一核心技术：RDD 算子。RDD 算子是 Spark 计算框架中定义的对 RDD 进行操作的各种函数，从 RDD 算子的功能角度来说，我们将 RDD 算子分为四类：创建算子、变换算子、行动算子和缓存算子。本章我们就对这 4 类中的主要算子进行详细说明，为了便于对众多算子的理解和掌握，我们还会通过实际用例展示这些算子的作用。

4.1 创建算子

创建 RDD 可以说是使用 Spark 处理和分析数据的第一步。创建 RDD 有两种方式，一种是将基于 Scala 的集合类型数据（例如 list 或 set 类型）分布到集群中生成 RDD，另一种是加载外部的数据源（例如本地文本文件或 HDFS 文件）生成 RDD。这两种方式都是通过 SparkContext 的接口函数提供的，其中，前者比较简单，仅包含两个函数：makeRDD 和 parallelize，而后面一种方式为了支持不同形式和不同格式的文件，提供了较多的函数。下面我们逐一说明。

4.1.1 基于集合类型数据创建 RDD

- **SparkContext. makeRDD——创建 RDD**

√ 算子函数格式：

makeRDD 算子有两个形式，一个可以指定分区数量，另一个可以指定分区所在的节点，下面我们分别进行说明。

```
- makeRDD[T](seq:Seq[T],numSlices:Int = defaultParallelism)(implicit
arg0:ClassTag[T]):RDD[T]
```

输入参数 seq 为一个数据集，numSlices 是分区的数量，如果不指定数量，将使用 Spark 配置中的 spark.default.parallelism 参数所生成的 defaultParallelism 数值，为默认的分区数量。函数执行后将 seq 指定的数据集分布到节点上形成 RDD，并返回生成的 RDD。

```
- makeRDD[T](seq:Seq[(T, Seq[String])])(implicit arg0: ClassTag[T]):RDD
[T]
```

输入参数 seq 为一个集合数据集，参数 String 序列指定了希望将该数据集产生的 RDD 分区希望放置的节点，这些节点可以使用 Spark 节点的主机名（Hostname）描述。算子执行后，集合数据 seq 将分布到这些节点上形成 RDD，并返回最终生成的 RDD。

√ 示例：

（注：以下示例可以在 Spark Shell 中执行，其中每行开始为语句的行标号，“scala >”后为输入的 Scala 语句，其他的为执行结果，本章之后的示例代码均为此格式。）

代码第 1 行将 1 到 6 构成的数组，生成 2 个分片的 RDD 赋给变量 rdd。代码第 2 行使用 collect 算子（后面我们会介绍），显示 rdd 变量的内容。代码第 3 行使用 partitions 算子显示 rdd 变量的分区信息。代码第 4 行生成数据序列 data，并指定分区 0（数据为 1 到 6 的序列）存放于 host1 和 host2，指定分区 1 到 10（数据为 7 到 10 的序列）存放于 host3。代码第 5 行使用生成的 data 序列产生新的 RDD 变量 rdd。代码第 6 行再次显示新的变量 rdd 中的内容。通过代码第 7 行和第 8 行，我们可以分别看到 rdd 的分区 0 和 1 所存储的位置。

makeRDD 示例代码

```
1:  scala> val rdd = sc.makeRDD(1 to 6, 2)
    rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at
makeRDD at <console>:21
2:  scala> rdd.collect
    res1: Array[Int] = Array(1, 2, 3, 4, 5, 6)
3:  scala> rdd.partitions
    res2: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.
ParallelCollectionPartition@6ba, org.apache.spark.rdd.
ParallelCollectionPartition@6bb)
4:  scala> val data = Seq((1 to 6, Seq("host1","host2")), (7 to 10, Seq
("host3")))
```

```
    data: Seq[(scala.collection.immutable.Range.Inclusive, Seq[String])]
=List((Range(1,2,3,4,5,6),List(host1,host2)), (Range(7,8,9,10),List
(host3)))
5:  scala> val rdd = sc.makeRDD(data)
    rdd: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.
Inclusive] = ParallelCollectionRDD[3]at makeRDD at <console>:23
6:  scala> rdd.collect
    res10: Array[scala.collection.immutable.Range.Inclusive] = Array
(Range(1,2,3,4,5,6), Range(7,8,9,10))
7:  scala> rdd.preferredLocations(rdd.partitions(0))
    res11: Seq[String] = List(host1, host2)
8:  scala> rdd.preferredLocations(rdd.partitions(1))
    res12: Seq[String] = List(host3)
```

- **SparkContext.parallelize——数据并行化生成 RDD**

√ 算子函数格式：

```
-parallelize[T](seq: Seq[T], numSlices: Int = defaultParallelism):RDD
```

将集合数据 seq 分布到节点上形成 RDD，并返回生成的 RDD。numSlices 是分区的数量，如果不指定数量，将使用 Spark 配置中的 spark.default.parallelism 参数所生成的 defaultParallelism 数值，为默认的分区数量。可以看到，parallelize 函数与上面的 makeRDD 作用类似，只是不能指定分区希望放置的节点。

√ 示例：

代码第 1 行将 1 到 6 构成的数组，生成 2 个分片的 RDD 赋给变量 rdd。代码第 2 行使用 collect 算子显示 rdd 变量的内容。代码第 3 行使用 partitions 算子显示 rdd 变量的分区信息。

parallelize 示例代码

```
1:  scala> val rdd = sc.parallelize(1 to 6, 2)
    rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
parallelize at <console>:21
2:  scala> rdd.collect
    res0: Array[Int] = Array(1,2,3,4,5,6)
3:  scala> rdd.partitions
    res1: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.
ParallelCollectionPartition@691, org. apache. spark. rdd.Parallel
CollectionPartition@ 692)
```

4.1.2 基于外部数据创建 RDD

Spark 支持两种从外部文件数据创建 RDD 的方式，一种是外部文本文件，另一种

是 Hadoop 文件系统中的各类文件格式。对于后者,由于 Hadoop 在 0.20 版本的前后分别有两种 InputFormat 定义(org.apache.hadoop.mapred 和 org.apache.hadoop.mapreduce.InputFormat),所以 Spark 的创建 RDD 操作接口也提供了对应的两套方式。由于使用旧 Hadoop API 的场景很少,仅在基于 Hadoop 0.20 之前版本安装的 Hadoop 集群环境中才需要。根据我们的了解,使用这种集群的环境已经非常少了,因此在这里我们就不再介绍此类接口,感兴趣的读者可以查阅 Spark 的官方 API 说明:https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext。

Spark 提供的基于外部数据创建 RDD 的算子主要有以下两类。

(1)基于文本文件创建 RDD。

- **Spark Context.textFile——基于文本文件创建 RDD**

√ 算子函数格式:

```
- textFile (path: String, minPartitions: Int = defaultMinSplits): RDD
[String]
```

从 HDFS、本地文件系统或者其他 Hadoop 支持的文件系统,按行读入指定路径下的文本文件,并返回生成的 RDD。path 是待读入的文本文件的路径。minSplits 是分区的数量,如果不特别指定,默认情况下将按照 min(defaultParallelism, 2)设置分区数量,其中 defaultParallelism 参数根据 Spark 配置中的 spark.default.parallelism 生成。

√ 示例:

textFile 示例代码

```
1:  scala>val textFile=sc.textFile("README.md")//读取 README.md 文件
    textFile: spark.RDD[String]=spark.MappedRDD@2ee9b6e3
2:  scala>textFile.count()//显示 rdd 的行数
    res0: Long=126
3:  scala>textFile.first()//显示 rdd 中第一行的内容
    res1: String=# Apache Spark
```

- **SparkContext.wholeTextFiles——基于一个目录下的全部文本文件创建 RDD**

√ 算子函数格式:

```
-wholeTextFiles(path:String,minPartitions:Int=defaultMinPartitions):
RDD[(String,String)]
```

从 HDFS、本地文件系统或者其他 Hadoop 支持的文件系统,按行读入指定目录下的所有文本文件,并返回生成的 RDD。path 是待读入的所有文本文件的所在目录。minPartitions 是分区的数量,如果不特别指定,默认情况下将按照 min(defaultParallelism, 2)设置分区数量,其中 defaultParallelism 参数根据 Spark 配置中的 spark. default. parallelism 生成。

√ 示例:

wholeTextFiles 示例代码

```
1:scala> val rdd = sc.wholeTextFiles("hdfs://data")
  rdd: org.apache.spark.rdd.RDD[(String, String)] = hdfs://data
  WholeTextFileRDD[0]at wholeTextFilesat <console>:12
2:scala> textFile.count()// 显示 rdd 的行数
  res0: Long = 126
3:scala> textFile.first() // 显示 rdd 中第一行的内容
  res1: String = # Apache Spark
```

(2) 基于新 Hadoop API 从 Hadoop 文件数据创建 RDD。

- **SparkContext. newAPIHadoopFile——基于 Hadoop 文件创建 RDD**

√ 算子函数格式:

newAPIHadoopFile 函数有两个形式,一个不可以设定 Hadoop 配置,另一个则可以设定 Hadoop 配置。

```
-newAPIHadoopFile[K, V, F <:InputFormat [K, V]](path: String)(implicit km:
ClassTag[K], vm: ClassTag[V],fm:ClassTag[F]):RDD[(K,V)]
```

从 HDFS 文件系统或者其他 Hadoop 支持的文件系统,读取 path 指定的输入文件路径,按照参数 F 指定的输入文件格式、参数 K 指定的 Key 类型和参数 V 指定的 Value 类型读取文件。

```
- newAPIHadoopFile[K, V, F <: InputFormat[K, V]](path: String, fClass:
Class[F], kClass: Class[K], vClass: Class[V], conf: Configuration =
hadoopConfiguration): RDD[(K, V)]
```

从 HDFS 文件系统或者其他 Hadoop 支持的文件系统,读取 path 指定的输入文件路径,按照参数 F 指定的输入文件格式、参数 K 指定的 Key 类型和参数 V 指定的 Value 类型读取文件,并可使用参数 conf 传入 Hadoop 配置。

√ 示例：

newAPIHadoopFile 示例代码

```
1:  scala > val rdd = sc.newAPIHadoopFile[LongWritable, Text, TextInput
Format]("README.md")
    rdd: org.apache.spark.rdd.RDD[(org.apache.hadoop.io.LongWritable,
org. apache. hadoop. io. Text)] = README. md NewHadoopRDD [1] at
newAPIHadoopFile at <console>:14
2:  scala>textFile.count()//显示 rdd 的行数
    res0: Long =126
3:  scala>textFile.first()//显示 rdd 中第一行的内容
    res1: String = # Apache Spark
```

- **SparkContext. newAPIHadoopRDD——基于 Hadoop 数据创建 RDD**

√ 算子函数格式：

```
- newAPIHadoopRDD[K, V, F <: InputFormat[K, V]](conf: Configuration =
hadoopConfiguration, fClass: Class[F], kClass: Class[K], vClass: Class
[V]): RDD[(K, V)]
```

参数 conf 指定 Hadoop 配置，参数 F 指定的输入数据格式、参数 K 指定的 Key 类型，参数 V 指定 Value 类型。与上面的“newAPIHadoopFile”接口不同，“newAPIHadoopRDD”没有文件路径的参数，因此这个接口通常用来读取诸如 HBase 这类数据库中的数据构建 RDD。

√ 示例：

下面的示例代码（注意，它们不是 scale shell 下可运行的脚本）展示了如何使用 newAPIHadoopRDD 从 HBase 表 user 中读取年龄大于 20 岁的用户信息，并统计人数的过程。

newAPIHadoopRDD 示例代码

```
1:  conf.set(TableInputFormat.INPUT_TABLE, "user")//设置使用的 HBase 表名
2:  val scan =new Scan()//设置过滤条件为年龄大于 20 岁
3:  scan.setFilter(new SingleColumnValueFilter("basic".getBytes,"age".
getBytes,CompareOp.GREATER_OR_EQUAL,Bytes.toBytes(18)))
4:  conf.set(TableInputFormat.SCAN,convertScanToString(scan))
5:  val usersRDD =sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],
    classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
    classOf[org.apache.hadoop.hbase.client.Result])//读取数据构建 RDD
6:  val count =usersRDD.count()//统计符合条件的用户数
7:  println("Users RDD Count:" + count)//输出结果
```

4.2 变换算子

变换(Transformation)算子就是对 RDD 进行操作的接口函数,其作用是将一个或多个 RDD 变换为新的 RDD。如果说 RDD 是 Spark 计算模式中的核心,那么变换类算子可以说是 Spark 核心中的核心。使用 Spark 进行数据计算,在利用创建算子生成 RDD 后,数据处理的算法设计和程序编写的最关键部分,就是利用变换算子对原始数据产生的 RDD 进行一步一步的变换,最终得到期望的计算结果。也因为其重要性,Spark 中提供了大量的变换类算子,这些算子都是 RDD 对象的接口函数。为了便于理解和掌握众多的变换算子,我们将它们分为两类:①对 Value 型 RDD 进行变换的算子;②对 Key/ Value 型 RDD(或称 PairRDD)进行变换的算子。同时,每个大类都按照算子可操作的 RDD 数量分为仅对一个 RDD 进行变换和对两个 RDD 进行变换的两个小类。下面我们就对它们进行详细并带有实例的说明。考虑到阅读本书的读者应该大多对 Java 比较了解,但对 Scala 并不一定很熟悉,因此我们在展示后面这些相对复杂的算子时选取的是算子的 Java 版本以便于理解。

4.2.1 对 Value 型 RDD 进行变换

对 Value 型 RDD 进行变换,可以通过 org. apache. spark. api. java. JavaRDD 类下的接口函数进行操作,本节我们对其中的一些主要函数进行带有实例的介绍。更多其他函数可以访问 Spark 的官方 API 查看:http:// spark. apache. org/ docs/ latest/ api/ scala/ index. html#org. apache. spark. api. java. JavaRDD。

4.2.1.1 对单个 Value 型的 RDD 进行变换

- **map——map 变换**

√ 算子函数格式:

```
-map[R](f:Function[T,R]):JavaRDD[R]
```

map 函数是 Hadoop 的 MapReduce 计算模式中耳熟能详的计算函数,因此我们选择从 map 算子开始介绍 Spark 的变换算子。与 Hadoop 的 Map 函数类似,map 算子也是将函数 f 作用于当前 RDD 的每个元素,形成一个新的 RDD。

√ 示例：

map 示例代码

```
1:  scala > val list = sc.parallelize (List (1,2,3,4),2)// 构建一个名为 list 的 RDD
    list: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0]at parallelize at <console>:21
2:  scala > list.map(x => x +1).collect //对 list 中的每个元素进行 +1 操作,形成新的 RDD
    res0: Array[Int] = Array(2, 3, 4, 5)
```

示例代码的变换过程可由图 4-1 所示，左侧 RDD（实线圆角矩形）中的每个元素经过函数 f 变换后，形成右侧新的 RDD。其中左侧虚线矩形代表 RDD 中的一个分区，我们可以看到 map 算子操作的 RDD 中每个分区会分别由函数 f 进行变换生成对应的一个新的分区。

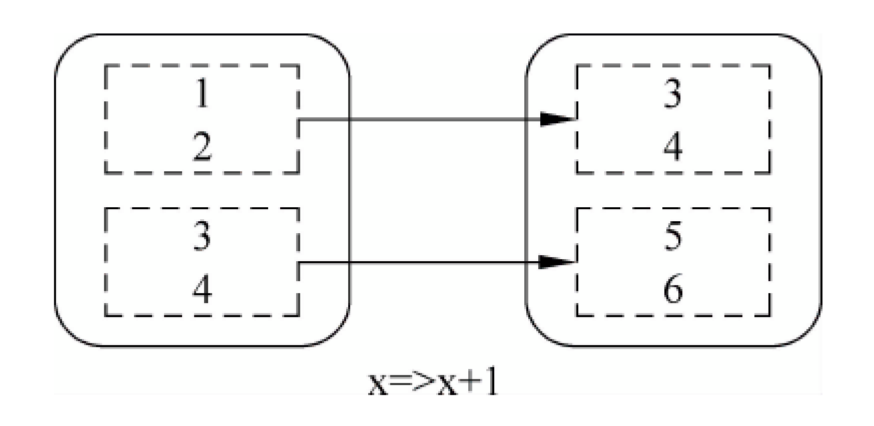


图 4-1 map 算子变换过程

- **coalesce——重新分区**

√ 算子函数格式：

```
- coalesce(numPartitions: Int, shuffle: Boolean): JavaRDD[T]
```

将当前 RDD 进行重新分区，生成一个以 numPartitions 参数指定的分区数存储的新 RDD。参数 shuffle 为 true 时在变换过程中进行 shuffle 操作，否则不进行 shuffle。

√ 示例：

coalesce 示例代码

```
1:  scala > val rdd = sc.parallelize(List(1,2,3,4,5,6,7,8),4)// 构建一个 4 个分区的 RDD
     rdd: org.apache.spark.rdd.RDD [Int] = ParallelCollectionRDD [5] at parallelize at <console>:12
2:  scala > rdd.partitions.length //显示 rdd 的分区数量
    res0: Int = 4
3:  scala > rdd.glom.collect //使用 glom 变换显示 rdd 存放在 4 个分区中的数据
```

```
    res1: Array[Array[Int]] = Array(Array(1, 2), Array(3, 4), Array(5, 6),
Array(7, 8))
4:  scala> val newRDD = rdd.coalesce(2, false) //将 rdd 变换为只有 2 个分区的
新 RDD
    newRDD: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[1]at coalesce at
<console>:14
5:  scala> newRDD.partitions.length //显示 newRdd 的分区数
    res2: Int = 2
6:  scala> newRDD.glom.collect //使用 glom 变换显示 newRDD 存放在 2 个分区中的
数据
    res1: Array[Array[Int]] = Array(Array(1, 2, 3, 4), Array(5, 6, 7, 8))
```

示例代码的变换过程如图 4-2 所示,左侧 RDD 为 4 个分区,经过变换后生成右侧为 2 个分区的新的 RDD。

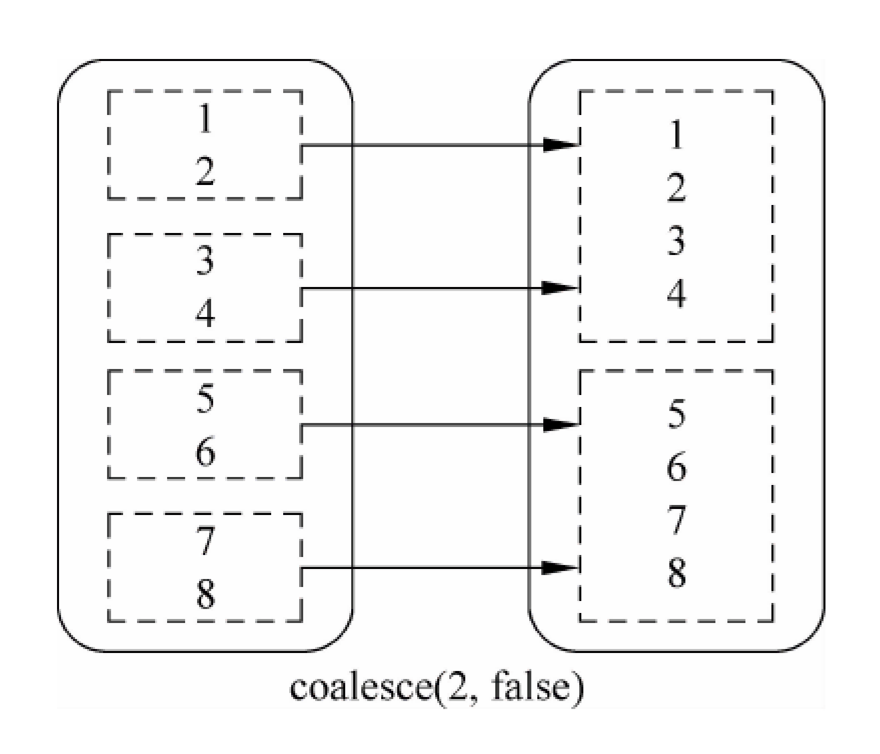


图 4-2 coalesce 算子变换过程

- **distinct——去重**

√ 算子函数格式:

makeRDD 去重变换有两个形式,一个不可以指定分区数量,另一个则可以指定分区数量。

```
-distinct(): JavaRDD[T]
```

返回原 RDD 中所有元素去重之后的结果,即原 RDD 中每个元素在新生成的 RDD 中只出现一次。

```
-distinct(numPartitions: Int): JavaRDD[T]
```

返回原 RDD 中所有元素去重之后的结果,即原 RDD 中每个元素在新生成的 RDD 中只出现一次,且新 RDD 的分区数等于 numPartitions。

√ 示例：

distinct 示例代码

```
1:  scala>val rdd=sc.parallelize(List(1,1,1,1,2,2,2,3,3,4),2)//生成由数字构成的 RDD
     rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at <console>:12
2:  scala>rdd.distinct.collect //显示 rdd 去重后的结果
    res0:Array[Int]=Array(4, 2, 1, 3)
```

示例代码的变换过程如图 4-3 所示，左侧原始 RDD 为 2 个分区，包含很多数字，其中有若干重复数字，经过 distinct 变换后，原 RDD 中的每个数字只出现一次，由于没有指定分区数，因此新生成 RDD 的分区数与原 RDD 相同。

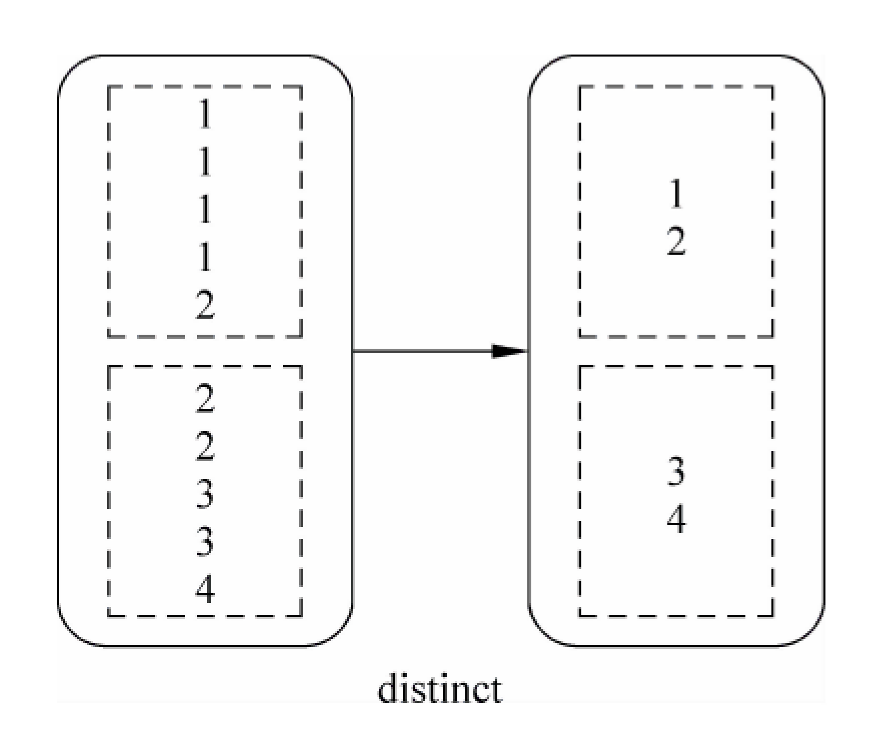


图 4-3　distinct 算子变换过程

- **filter——过滤**

√ 算子函数格式：

```
-filter(f:Function[T, Boolean]):JavaRDD[T]
```

原 RDD 中的所有元素，通过输入参数 f 指定的过滤函数进行判别，使过滤函数返回为 true 的所有元素构成一个新的 RDD。

√ 示例：

filter 示例代码

```
1:  scala>val rdd=sc.parallelize(0 to 9,2)//生成由数字 0-9 序列构成的 RDD
     rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at parallelize at <console>:12
2:  scala>val filteredRDD=rdd.filter(_% 2==0)//从 rdd 中挑出能被 2 整除
```

```
的数构成新 RDD
    res0: Array[Int] = Array(0, 2, 4, 6, 8)
```

示例代码的变换过程如图 4-4 所示,左侧原始 RDD 由数字 0-9 的序列构成,指定的过滤函数为挑选能被 2 整除的数,从而构成右侧新的 RDD。

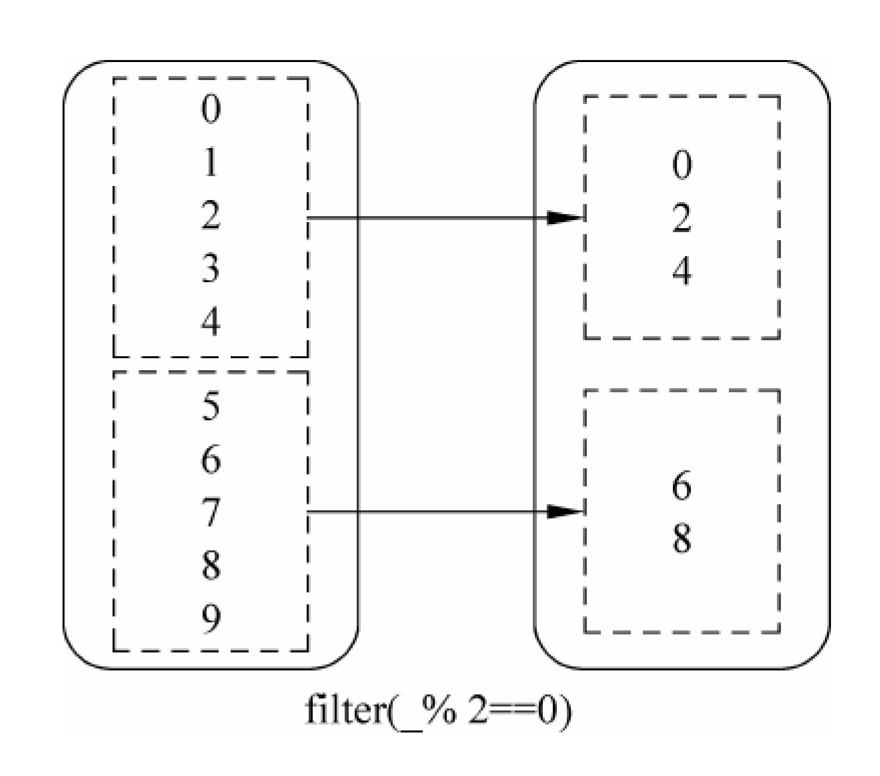


图 4-4 filter 算子变换过程

- **flatMap——flatMap 变换**

√ 算子函数格式:

```
-flatMap[U](f: FlatMapFunction[T, U]): JavaRDD[U]
```

在前面我们已经了解到 map 变换是对原 RDD 中的每个元素进行一对一变换生成新 RDD,而 flatMap 不同的地方在于,它是对原 RDD 中的每个元素用指定函数 f 进行一对多(这也是 flat 前缀的由来)的变换,然后将转换后的结果汇聚生成新 RDD。

√ 示例:

flatMap 示例代码

```
1: scala> val rdd = sc.parallelize(0 to 3,1) //生成由 0 - 3 序列构成的 RDD
    rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[17] at parallelize at <console>:21
2: scala> val flatMappedRDD = rdd.flatMap(x => 0 to x) //使用 flatMap 将每个原始变换为一个序列
    flatMappedRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[18] at flatMap at <console>:23
3: scala> flatMappedRDD.collect // 显示新的 RDD
    res0: Array[Int] = Array(0, 0, 1, 0, 1, 2, 0, 1, 2, 3)
```

示例代码的变换过程如图 4-5 所示,左侧原始 RDD 由数字 0-3 的序列构成,flatMap 指定的变换函数为将每个元素被转化为 0 到自身这个区间范围内的一个列表,

例如2变换为{0,1,2},最终将所有列表合并生成右侧新的RDD。

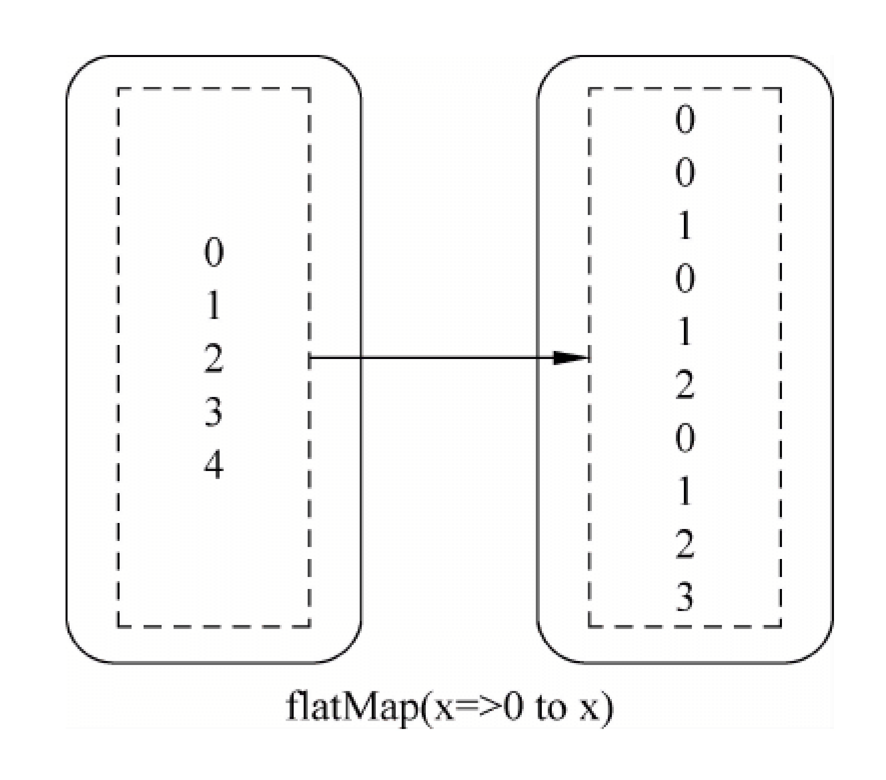


图4-5 flatMap算子变换过程

- **pipe——调用Shell命令**

√ 算子函数格式:

```
-pipe(command: String): JavaRDD[String]
```

在Linux操作系统中,提供了很多能对数据进行处理的shell命令。为了利用上这些命令的能力,Spark提供了pipe变换。利用pipe变换,可以对原RDD的每个分区执行由command参数指定的shell命令,并生成新的RDD。

√ 示例:

pipe示例代码

```
1:  scala>val rdd=sc.parallelize(0 to 7,4)//生成由0-9的序列构成的RDD,存放在5个分区中
    rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at parallelize at <console>:12
2:  scala>rdd.glom.collect//显示每个分区的数据
    res0: Array[Array[Int]] = Array(Array(0, 1), Array(2, 3), Array(4, 5), Array(6, 7))
3:  scala>rdd.pipe("head -n 1").collect //提取每个分区的第1个元素生成新的RDD
    res1: Array[String] = Array(0, 2, 4, 6)
```

示例代码的变换过程如图4-6所示,左侧原始RDD由数字0~7的序列构成,并分为4个分区存放。然后使用pipe变换调用Linux的head命令,提取每个分区中的第1个元素构成一个新的RDD。

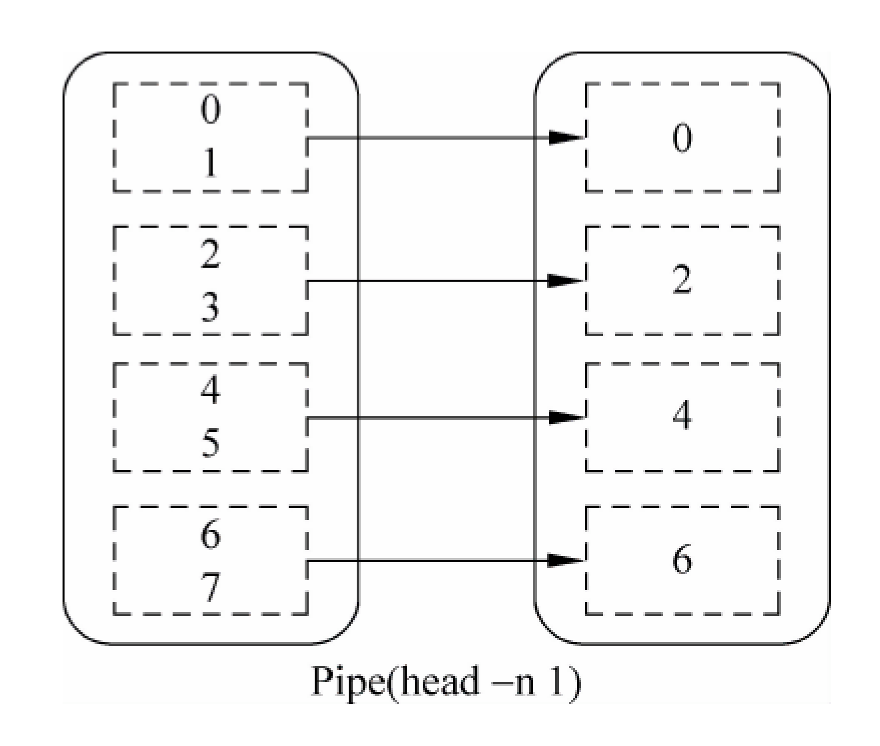


图 4-6 pipe 算子变换过程

- **sample——抽样**

√ 算子函数格式:

```
-sample(withReplacement: Boolean, fraction: Double, seed: Long): JavaRDD
[T]
```

对原始 RDD 中的元素进行随机抽样,抽样后产生的元素集合构成新的 RDD。参数 fraction 指定新集合中元素的数量占原始集合的比例。抽样时的随机数种子由 seed 指定。参数 withReplacement 为 false 时,抽样方式为不放回抽样。参数 withReplacement 为 true 时,抽样方式为放回抽样。

√ 示例:

sample 示例代码

```
1:  scala> val rdd = sc.parallelize(0 to 9,1) //生成由 0 - 9 的序列构成的 RDD
     rdd: org.apache.spark.rdd.RDD [Int] = ParallelCollectionRDD [5] at
parallelize at <console>:21
2:  scala> rdd.sample(false,0.5).collect //不放回抽样一半比例的元素生成新
的 RDD
    res4: Array[Int] = Array(0, 1, 2, 3, 4, 7)
3:  rdd.sample(false,0.5).collect //再次不放回抽样一半比例的元素生成新的 RDD
    res7: Array[Int] = Array(0, 1, 3, 6, 8)
4:  scala> rdd.sample(false,0.8).collect //不放回抽样 80% 比例的元素生成新的
RDD
    res8: Array[Int] = Array(0, 1, 2, 5, 6, 8, 9)
5:  scala> rdd.sample(true,0.5).collect //放回抽样一半比例的元素生成新的 RDD
    res9: Array[Int] = Array(0, 2, 3, 4, 4, 6, 7, 9)
```

示例代码的第一次(第 2 行)变换过程如图 4-7(a)所示,左侧原始 RDD 由数字 0 ~ 9 的序列构成,然后使用 sample 变换随机提取一半数量的元素构成新的 RDD,且采

用不放回抽样的形式,此时生成的结果为右侧6个元素构成的RDD。而在之后的代码中我们还试验了同样条件再次随机抽样、比例更大、放回抽样的方式,可以看到其结果各有不同,如图4-7(b)所示。

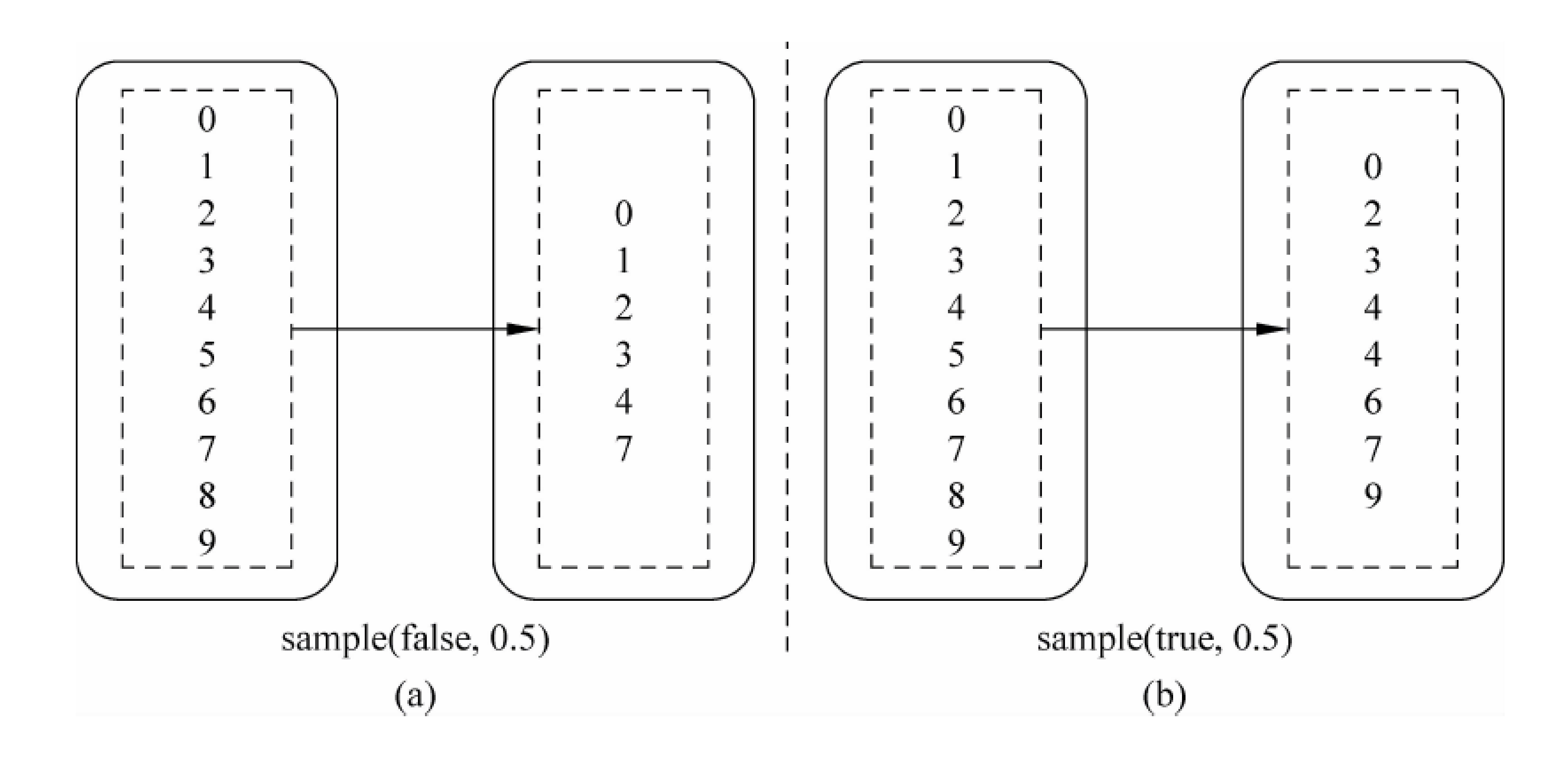


图4-7 sample算子变换过程

- **sortBy——排序**

√ 算子函数格式:

```
- sortBy[S](f: Function[T, S], ascending: Boolean, numPartitions: Int):
JavaRDD[T]
```

对原RDD中的元素按照函数f指定的规则进行排序,并可使用参数ascending指定按照升序或者降序进行排列,排序后的结果生成新的RDD,新RDD的分区数量可以由参数numPartitions指定,默认采用与原RDD相同的分区数。

√ 示例:

sortBy示例代码

```
1:  scala> val rdd = sc.parallelize(List(2,1,4,3),1)//生成原始RDD
    rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18]at
parallelize at <console>:12
2:  scala> rdd.sortBy(x => x,true).collect//对原始RDD中的元素进行升序排列后的结果
    res0: Array[Int] = Array(1, 2, 3, 4)
3:  scala> rdd.sortBy(x => x,false).collect //对原始RDD中的元素进行降序排列后的结果
    res1:  Array[Int] = Array(4, 3, 2, 1)
```

示例代码的第 1 次变换过程(第 2 行)如图 4-8 所示,左侧原始 RDD 由数字 2、1、4、3 构成,然后使用 sortBy 变换使用元素的值进行升序排列,生成的新 RDD 由右侧有序数字序列构成。在示例代码的第 3 行我们还试验了降序排列的结果。

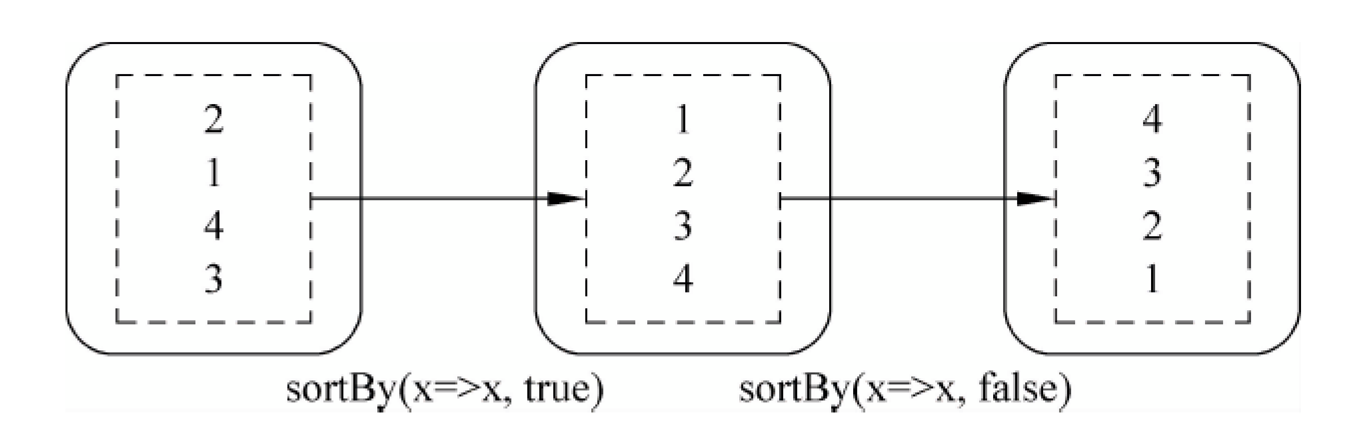


图 4-8 sortBy 算子变换过程

4.2.1.2 对两个 Value 型 RDD 进行变换

- **cartesian——笛卡尔积**

√ 算子函数格式:

```
-cartesian[U](other: JavaRDDLike[U, _]): JavaPairRDD[T, U]
```

输入参数为另一个 RDD,返回两个 RDD 中所有元素的笛卡尔积,即生成由当前 RDD 的所有元素与输入参数 RDD 的所有元素两两组合构成的所有可能的有序对集合。

√ 示例:

cartesian 示例代码

```
1:  scala> val rdd_1 = sc.parallelize(List("a","b","c"),1)
    rdd_1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0]at
parallelize at <console>:12
2:  scala> val rdd_2 = sc.parallelize(List(1,2,3),1)
    rdd_2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1]at
parallelize at <console>:12
3:  scala> rdd_1.cartesian(rdd_2).collect
    res0: Array[(String, Int)] = Array((a,1), (a,2), (a,3), (b,1), (b,2), (b,
3), (c,1), (c,2), (c,3))
```

示例代码的第 1 次变换过程如图 4-9 所示,左侧上面的原始 RDD 由字母 A、B、C 构成,下面的输入参数 RDD 由数字 1、2、3 构成,使用 cartesian 变换后得到两个 RDD 的所有元素构成的有序对集合。

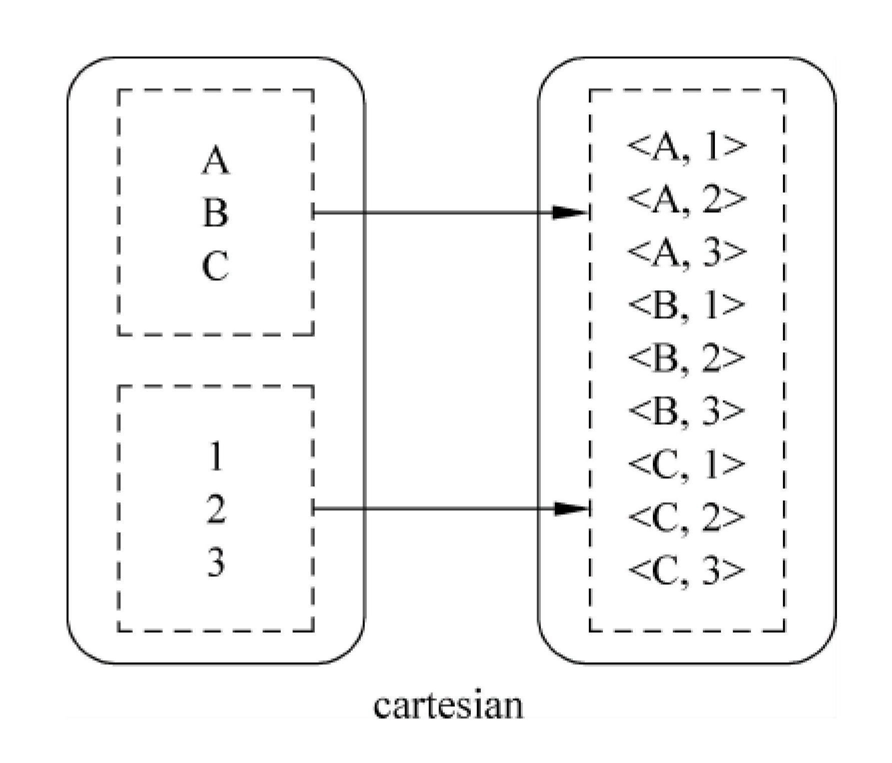


图 4-9 cartesian 算子变换过程

- **intersection——交集**

√ 算子函数格式：

```
-intersection(other: JavaRDD[T]): JavaRDD[T]
```

输入参数为另一个 RDD，返回两个 RDD 中所有元素的交集，即生成由同时在两个 RDD 中存在的元素集合构成的新 RDD，同时可以使用参数 numPartitions 指定新 RDD 的分区数。

√ 示例：

intersection 示例代码

```
1:  scala> val rdd_1 = sc.parallelize(List(1,2,3,4),1)//构建原始 RDD
    rdd_1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
parallelize at <console>:21
2:  scala> val rdd_2 = sc.parallelize(List(2,3,4,5),1) //构建输入参数 RDD
    rdd_2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at
parallelize at <console>:21
3:  scala> rdd_1.intersection(rdd_2,1).collect //求两个 RDD 的交集
    res0: Array[Int] = Array(4, 3, 2)
```

示例代码的变换过程如图 4-10 所示，左侧上面原始 RDD 由数字 1、2、3、4 构成，下面的输入参数 RDD 由数字 2、3、4、5 构成，使用 intersection 变换后得到两个 RDD 的共同元素 2、3、4 构成的新 RDD。

- **subtract——补集**

√ 算子函数格式：

```
-subtract(other: JavaRDD[T], numPartitions: Int): JavaRDD[T]
```

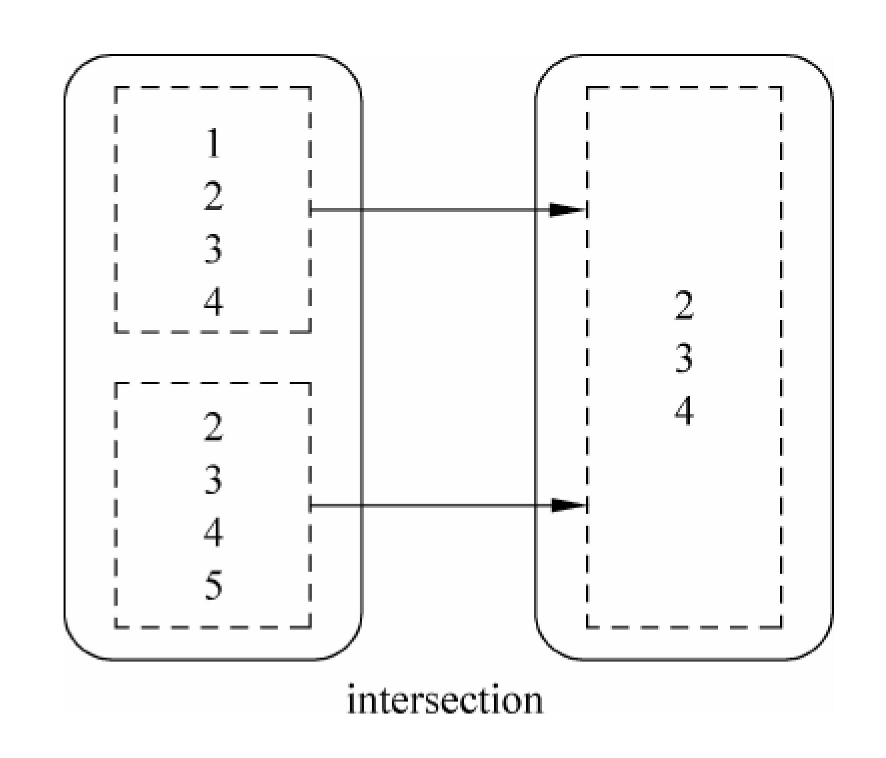


图 4-10 intersection 算子变换过程

输入参数为另一个 RDD,返回原始 RDD 与输入参数 RDD 的补集,即生成由在原始 RDD 中而不在输入参数 RDD 中的元素集合构成的新 RDD,同时可以使用参数 numPartitions 指定新 RDD 的分区数。

√ 示例:

subtract 示例代码

```
1:  scala> val rdd_1 = sc.parallelize(0 to 5,1)
    rdd_1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at
parallelize at <console>:12
2:  scala> val rdd_2 = sc.parallelize(0 to 2,1)
    rdd_2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at
parallelize at <console>:12
3:  scala> rdd_1.subtract(rdd_2).collect
    res0: Array[Int] = Array(3, 4, 5)
```

示例代码的变换过程如图 4-11 所示,左侧上面原始 RDD 由数字 0 ~5 构成,下面的输入参数 RDD 由数字 0 ~2 构成,使用 subtract 变换后得到在原始 RDD 中而不在输入参数 RDD 中的元素 3 ~5 构成的新 RDD。

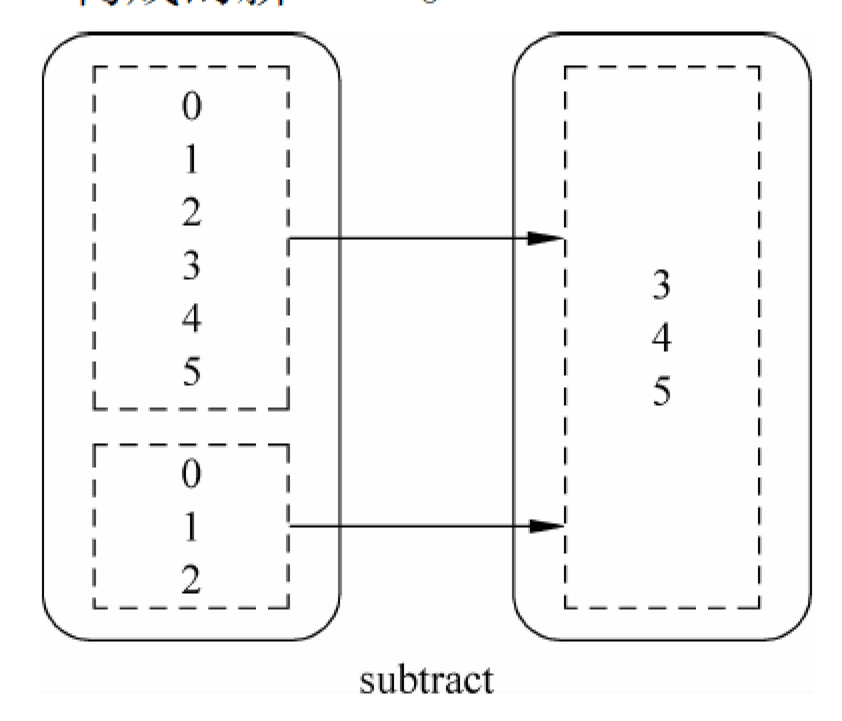


图 4-11 subtract 算子变换过程

- **union——并集**

√ 算子函数格式：

```
-union(other: JavaRDD[T]): JavaRDD[T]
```

输入参数为另一个 RDD，返回原始 RDD 与输入参数 RDD 的并集，即生成原始 RDD 和输入参数 RDD 中的所有元素构成的集合，如果两个 RDD 中有重复元素，则这些元素会多次出现。如果需要去除重复原始，可以对新生成的 RDD 使用 distinct 变换。

√ 示例：

union 示例代码

```
1:  scala> val rdd_1 = sc.parallelize(0 to 5,1)//构建原始 RDD
    rdd_1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9]at
parallelize at <console>:21
2:  scala> val rdd_2 = sc.parallelize(4 to 6,1) //构建输入参数 RDD
    rdd_2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10]at
parallelize at <console>:21
3:  scala> rdd_1.union(rdd_2).collect //生成两个 RDD 的并集
    res7: Array[Int] = Array(0, 1, 2, 3, 4, 5, 4, 5, 6)
4:  scala> rdd_1.union(rdd_2).distinct().collect //生成两个 RDD 的并集并
去重
    res8: Array[Int] = Array(4, 0, 6, 2, 1, 3, 5)
```

示例代码的变换过程如图 4-12 所示，左侧上面原始 RDD 由数字 0～5 构成，下面的输入参数 RDD 由数字 4～6 构成，使用 union 变换后得到由两个 RDD 中的全部元素构成的新 RDD，可以看到在两个 RDD 中都存在的数字会重复出现。在第 4 行代码中我们使用 distinct 去重变换，则可以看到结果中不再有重复元素。

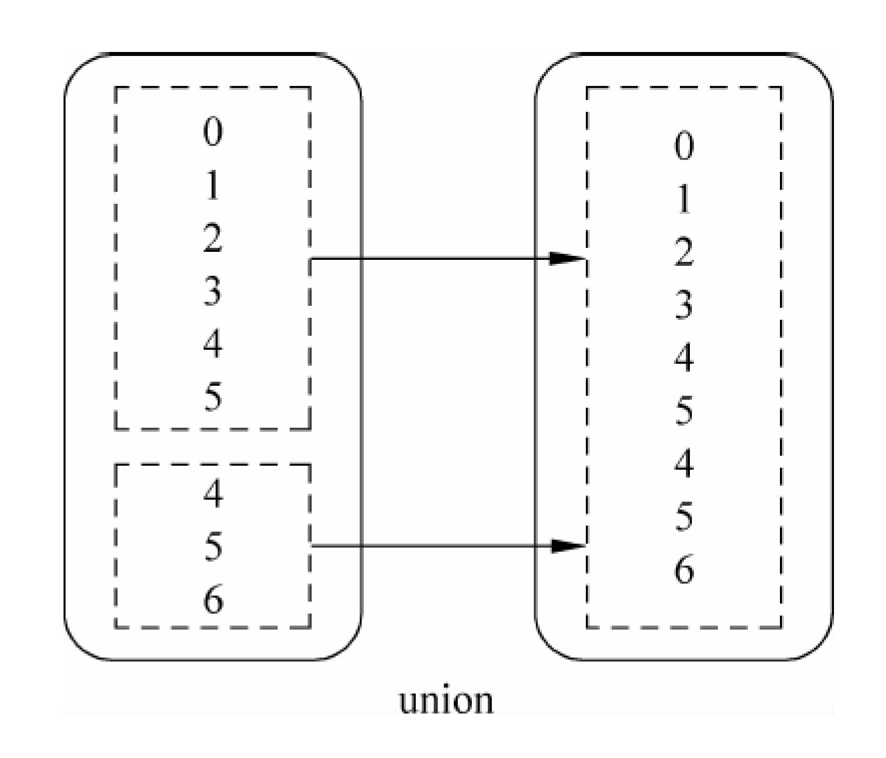


图 4-12 union 算子变换过程

- **zip——联结**

√ 算子函数格式：

```
-zip[U](other: JavaRDDLike[U, _]): JavaPairRDD[T, U]
```

输入参数为另一个 RDD，zip 变换生成由原始 RDD 的值为 Key、输入参数 RDD 的值为 Value 依次配对构成的所有 Key/ Value 对，并返回这些 Key/ Value 对集合构成的新 RDD。

√ 示例：

zip 示例代码

```
1:  scala> val rdd_1 = sc.parallelize(0 to 4,1) //构建原始 RDD
    rdd_1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[19]at
parallelize at <console>:21
2:  scala> val rdd_2 = sc.parallelize(5 to 9,1) //构建输入参数 RDD
    rdd_2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20]at
parallelize at <console>:21
3:  scala> rdd_1.zip(rdd_2).collect //对两个 RDD 进行联结
    res5: Array[(Int, Int)] = Array((0,5), (1,6), (2,7), (3,8), (4,9))
```

示例代码的变换过程如图 4-13 所示，左侧上面原始 RDD 由数字 0 ~4 构成，下面的输入参数 RDD 由数字 5 ~9 构成，使用 zip 变换后得到由两个 RDD 中的元素依次配对构成的 Key/ Value 对形成的新 RDD。

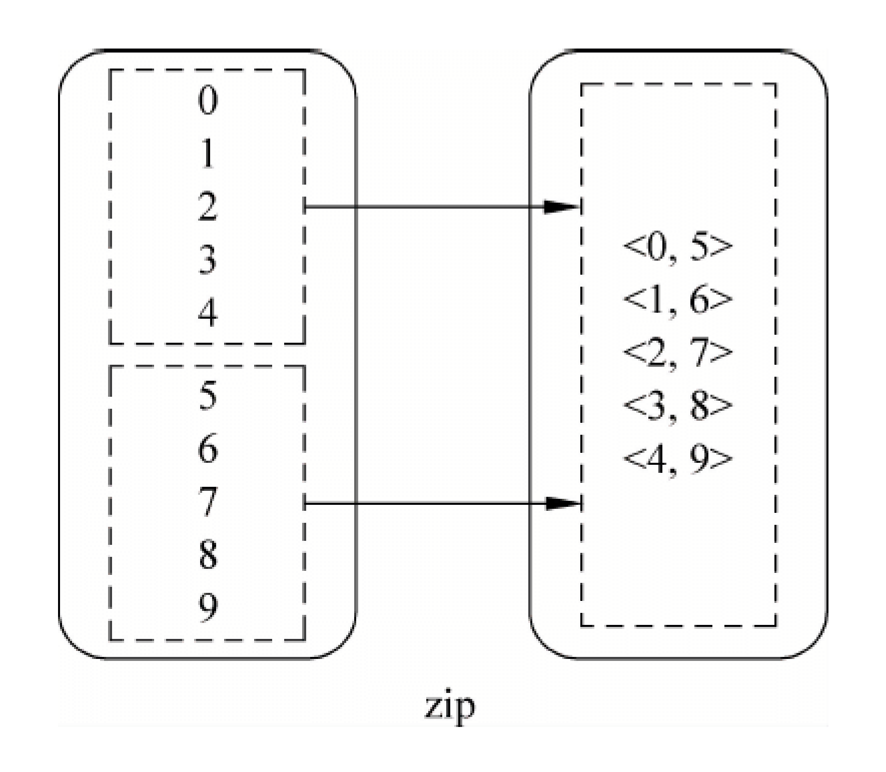


图 4-13 zip 算子变换过程

4.2.2 对 Key/ Value 型 RDD 进行变换

对 Key/ Value 型 RDD 进行变换，可以通过 org. apache. spark. api. java. JavaPairRDD 类下的接口函数进行操作，本节我们对其中的一些主要函数进行带有实

例的介绍。更多其他函数可以访问：https:// spark. apache. org/ docs/ latest/ api/ scala/ index. html#org. apache. spark. api. java. JavaPairRDD。

4.2.2.1 创建 Key/ Value 型 RDD

在处理 Key/ Value 型 RDD 之前，我们需要先创建这种类型的数据。创建 Key/ Value 型 RDD 的方法有很多，其基本思路都是构建由 Key 和 Value 构成的数据对，然后使用 RDD 变换函数创建对应的 RDD，这里我们用已经介绍过的简单易理解的 map 变换和新的 keyBy 变换两种方式创建 Key/ Value 型的 RDD。

- **使用 map 创建 Key/ Value 型 RDD**

使用 map 创建 Key/Value 型 RDD 示例代码

```
1:  scala > val words = sc.parallelize(List("apple","banana","berry","cherry","cumquat","haw"),1)//构建原始 RDD
    words: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0]at parallelize at <console>:21
2:  scala>words.collect //显示原始 RDD
    res0: Array[String] = Array(apple, banana, berry, cherry, cumquat, haw)
3:  scala>val pairs = words.map(x =>(x(0),x)) //使用 map 变换生成 Key/Value 型 RDD
    pairs: org.apache.spark.rdd.RDD[(Char, String)] = MapPartitionsRDD[1] at map at <console>:23
4:  scala>pairs.collect //显示生成的 Key/Value 型 RDD
    res1:  Array[(Char, String)] = Array((a,apple), (b,banana), (b,berry), (c,cherry), (c,cumquat), (h,haw))
```

示例代码的变换过程如图 4-14 所示，其中原始 RDD 中的元素为长度不等的单词，map 变换中将每个单词的第一个字母作为 Key 值，然后与对应单词一起构成键值对形成新的 Key/ Value 型 RDD。

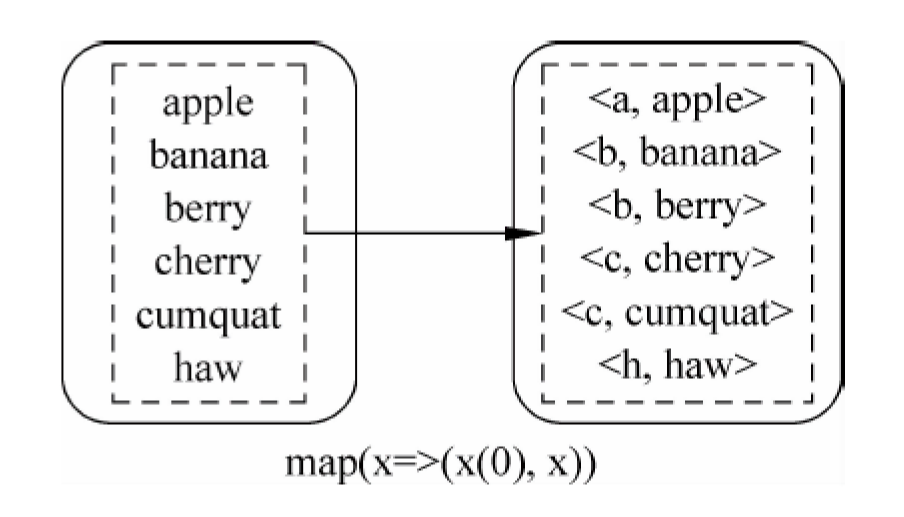


图 4-14 使用 map 创建 Key/ Value 型 RDD

- **使用 keyBy 创建 Key/ Value 型 RDD**

使用 keyBy 创建 Key/Value 型 RDD 示例代码

```
1: scala > val words = sc.parallelize (List ("apple","banana","berry",
"cherry","cumquat","haw"),1) //构建原始 RDD
    words: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[13]
at parallelize at <console>:12
2: scala > words.collect //显示原始 RDD
    res0: Array[String] = Array(apple, banana, berry, cherry, cumquat, haw)
3: scala > val pairs = words.keyBy(_.length) //使用 keyBy 变换生成 Key/Value
型 RDD
    pairs: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[14] at
keyBy at <console>:14
4: scala > pairs.collect //显示生成的 Key/Value 型 RDD
    res1:  Array[(Int, String)] = Array((5,apple), (6,banana), (5,berry),
(6,cherry), (7,cumquat), (3,haw))
```

keyBy 变换(keyBy[U](f: Function[T, U]): JavaPairRDD[U, T])的输入参数为构建 Key 的函数,此函数将作用于原 RDD 中的每个元素,然后与对应元素的原始值共同构成键值对,这些键值对集合构成新的 Key/ Value 型 RDD。示例代码的变换过程如图 4-15 所示,其中原始 RDD 中的元素为长度不等的单词,keyBy 变换中传递的函数为将每个单词的长度作为 Key 值,然后与对应单词一起构成键值对形成新的 Key/ Value 型 RDD。

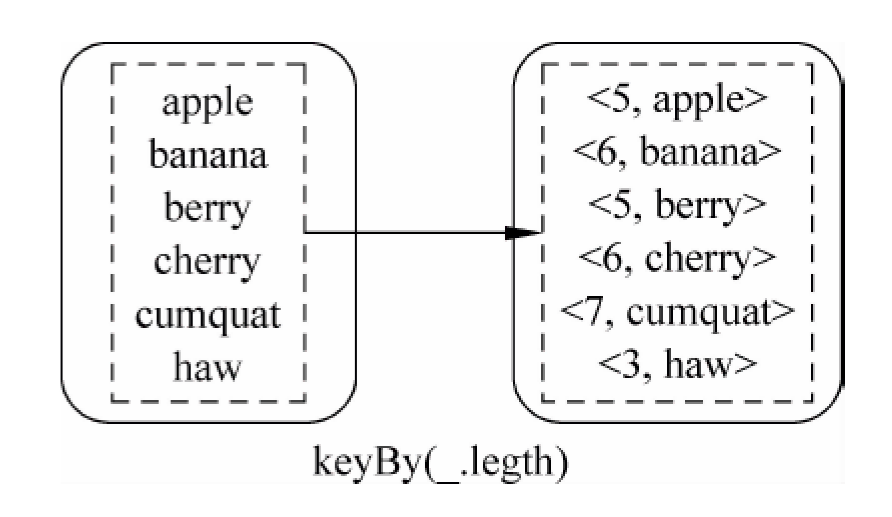


图 4-15 使用 keyBy 创建 Key/ Value 型 RDD

4.2.2.2 对单个 Key-Value 型 RDD 进行变换

- **combineByKey——按 Key 聚合**

√ 算子函数格式:

```
- combineByKey[C](createCombiner: Function[V, C], mergeValue: Function2
[C, V, C], mergeCombiners: Function2[C, C, C]): JavaPairRDD[K, C]
```

将 Key/ Value 型的 RDD 按 Key 进行聚合(Combine)计算,该变换的计算过程是将 RDD 中的 <Key, Value> 对依次输入该变换,然后使用输入参数中的 3 个函数进行如下计算。

- createCombiner：在遍历 RDD 的一个分区时,对于依次处理的每个 <Key, Value> 对,如果 Key 是第一次出现,则触发执行 createCombiner 函数,将 <Key, Value> 中的 Value 转换由函数指定的数据类型 C。
- mergeValue：对于依次处理的每个 <Key, value> 对,如果 Key 不是第一次出现,则触发执行 mergeValue 函数,将前一次计算获得的 C(可能是第一次 Key 到达时由 createCombiner 函数生成,也可能是后续 Key 到达时由 mergeValue 函数生成)与当前到达的 <Key, value> 对中的 Value 用 mergeValue 函数计算,得到新的 C。
- mergeCombiners：我们都知道,Spark 是一个分布式计算环境,在执行聚合变换时,RDD 可能分布在不同的计算节点进行变换。因此,要获得整体数据的聚合结果,最后还需要将分散的相同 Key 值的数据汇集到一起运算,这就是 mergeCombiners 的作用。在前面两个函数计算获得的所有的 C,都要经过 mergeCombiners 汇聚成最终结果的 C。

√ 示例：

combineByKey 示例代码

```
1:  scala > val pair = sc.parallelize(List(("fruit","Apple"),("fruit","Banana"),("vegetable","Cucumber"), ("fruit","Cherry"),("vegetable","Bean"), ("vegetable","Pepper")),2) //生成原始 RDD
    pair: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[10]at parallelize at <console>:12
2:  scala > val combinedPair = pair.combineByKey(List(_), (x:List[String], y:String) => y :: x, (x:List[String], y:List[String]) => x ::: y) //进 CombineByKey 变换
    combinedPair: org.apache.spark.rdd.RDD[(String, List[String])] = ShuffledRDD[1]at combineByKey at <console>:14
3:  scala > combinedPair.collect //输出变换结果
    res0: Array[(String, List[String])] = Array((fruit,List(Cherry, Banana, Apple)), (vegetable,List(Pepper, Cucumber, Bean)))
```

示例代码中,输入 RDD 为 Key/ Value 型的 RDD,Key 为物品的类型名称,例如水果(fruit)或蔬菜(vegetable),Value 为具体的物品名称,例如苹果(Apple)或香蕉

(Banana)。示例代码的功能是将属于同一类的物品汇聚到一起形成一个 List。其关键在于第 2 行代码中指定的 3 个函数：

－指定的 createCombiner 函数为 List(_)，其作用是在某类物品的第一个物品出现时，将该物品的名称放入一个 List 中，例如 <fruit, Apple> 到达时会生成 List(Apple)。

－指定的 mergeValue 函数为(x:List[String], y:String) = > y :: x，其作用是将不是第一次出现的类型的物品名称，放入已生成的 List 中，例如 <fruit, Banana> 到达时会生成 List(Apple, Banana)。

－指定的 mergeCombiners 函数为(x:List[String], y:List[String]) = > x ::: y，其作用是将前两个函数生成的某物品类别的所有列表合并为一个列表。

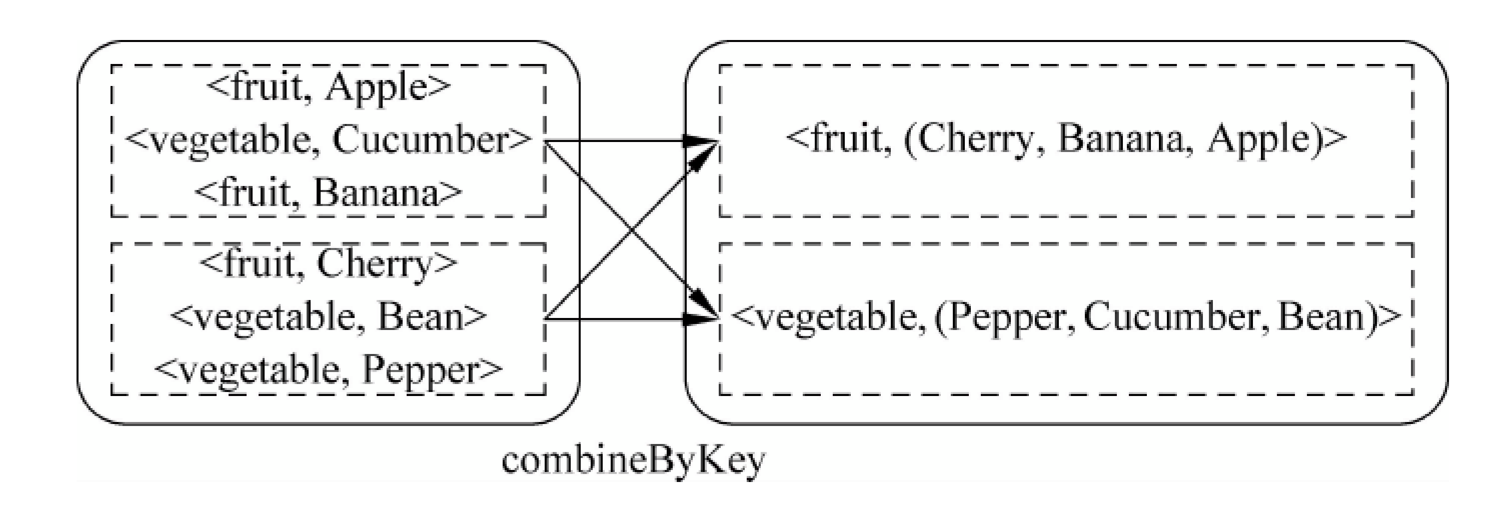


图 4-16　combineByKey 算子变换过程

- **flatMapValues——对所有 Value 进行 flatMap**

√ 算子函数格式：

```
- flatMapValues[U](f: Function[V, Iterable[U]]): JavaPairRDD[K, U]
```

对 Key/ value 型 RDD 中的所有 Value 值进行 flatMap 操作。

√ 示例：

flatMapValues 示例代码

```
1:  scala> val rdd = sc.parallelize(List("a","boy"),1).keyBy(_.length)
    rdd: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[1]at keyBy at
<console>:12
2:  scala> rdd.collect
    res0: Array[(Int, String)] = Array((1,a), (3,boy))
3:  scala> rdd.flatMapValues(x => "*" + x + "*").collect
    res1: Array[(Int, Char)] = Array((1,*), (1,a), (1,*), (3,*), (3,b), (3,
o), (3,y), (3,*))
```

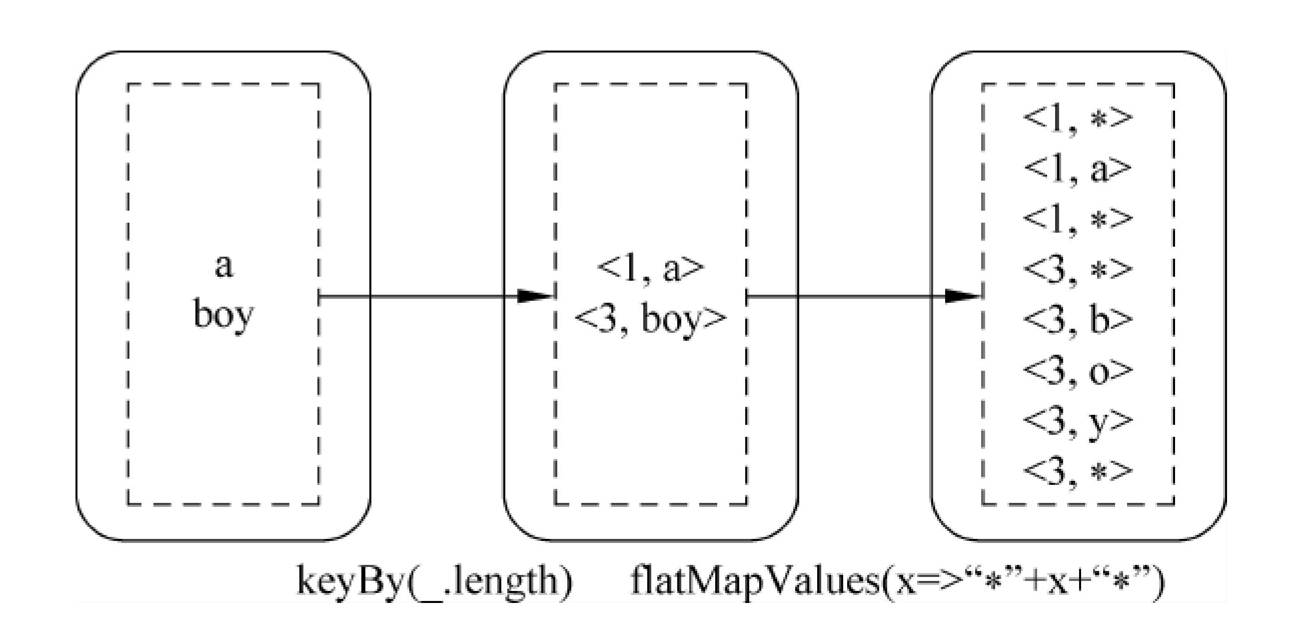


图 4-17 flatMapValues 算子变换过程

- **groupByKey——按 Key 汇聚**

√ 算子函数格式:

```
-groupByKey():JavaPairRDD[K, Iterable[V]]
```

将 Key/ Value 型 RDD 中的元素按 Key 值进行汇聚,Key 值相同的 Value 值合并在一个序列中,所有 Key 值的序列构成新的 RDD。需要注意的是,groupByKey 变换对每个分区进行操作后的输出不会进行合并,因此整个变换的开销是非常大的,应该尽量避免使用,尽量使用可以完成类似运算的 aggregateByKey 或 reduceByKey。groupByKey 变换还有两个变型,分别为可以指定分区数量的 groupByKey(numPartitions: Int)和可以指定分区类的 groupByKey(partitioner: Partitioner)。

√ 示例:

groupByKey 示例代码

```
1:  scala>val pairs=sc.parallelize(List(("fruit","Apple"),
("vegetable","Cucumber") ("fruit","Cherry"), ("vegetable","Bean"),
("fruit","Banana"),("vegetable","Pepper")),2) //生成原始 RDD
    pairs: org.apache.spark.rdd.RDD[(String, String)]=Parallel
CollectionRDD[5]at parallelize at <console>:12
2:  scala>pairs.groupByKey.collect //进行 groupByKey 变换
    res0: Array[(String, Iterable[String])]=Array((fruit,CompactBuffer
(Apple, Banana, Cherry)), (vegetable, CompactBuffer (bean, cucumber,
pepper)))
```

在示例代码中,输入 RDD 为 Key/ Value 格式的 RDD,Key 为物品的类型名称,例如水果(fruit)或蔬菜(vegetable),Value 为具体的物品名称,例如苹果(Apple)或香蕉(Banana)。通过 groupByKey 变换将类型相同的物品名称合并为一个 List。

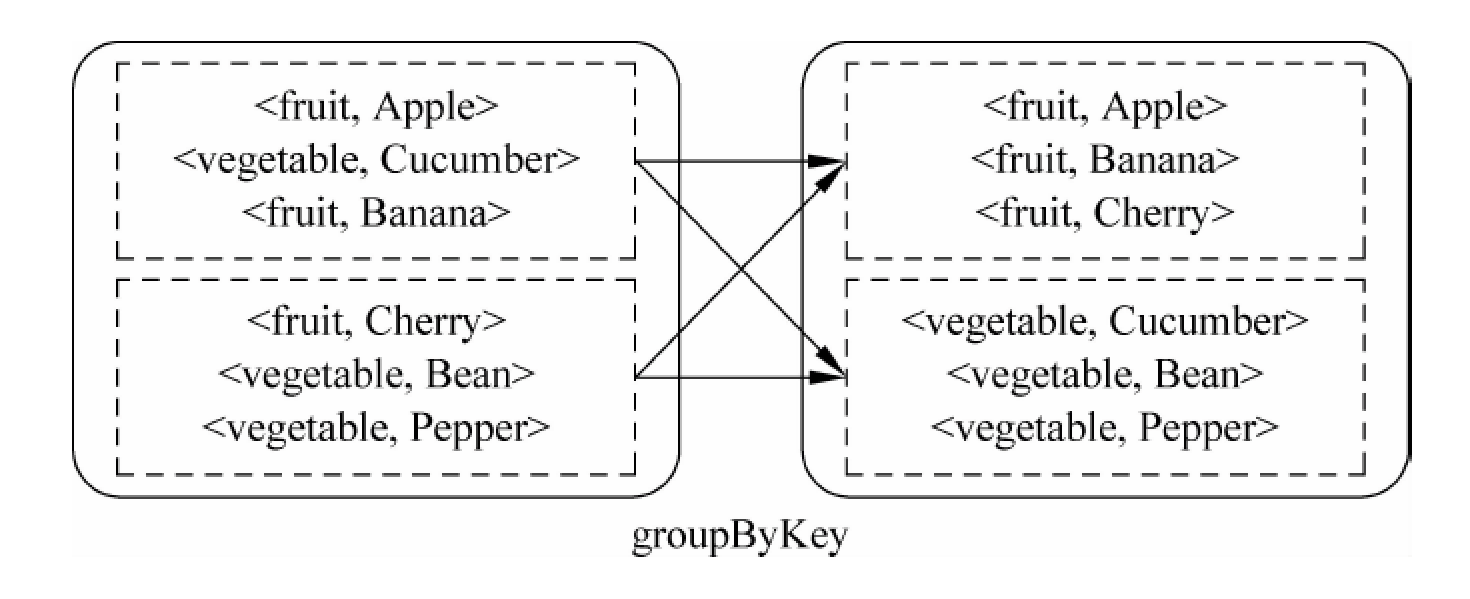


图 4-18 groupByKey 算子变换过程

- **keys——提取 Key**

√ 算子函数格式：

```
-keys(): JavaRDD[K]
```

将 Key/ Value 型 RDD 中的元素的 Key 提取出来，所有 Key 值构成一个序列形成新的 RDD。

√ 示例：

keys 示例代码

```
1:  scala > val pairs = sc.parallelize(List("apple","banana","berry",
"cherry","cumquat","haw"),1).keyBy(_.length) //构建原始 RDD
    pairs: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[16]at keyBy
at <console>:12
2:  scala>pairs.keys.collect //提取单词长度
    res0: Array[Int] = Array(5, 6, 5, 6, 7, 3)
```

示例代码中，先生成一个由单词长度与单词构成的 Key/ Value 型 RDD，然后使用 Keys 变换得到所有单词的长度。

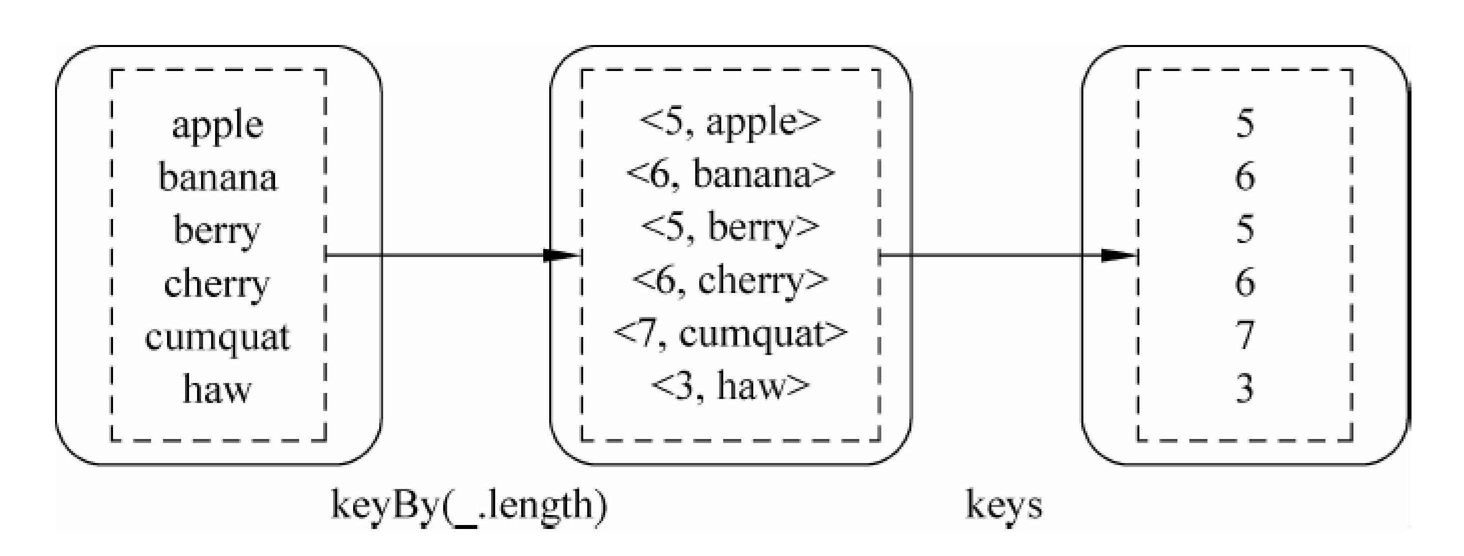


图 4-19 keys 算子变换过程

- **mapValues——对 Value 值进行变换**

√ 算子函数格式：

```
-mapValues[U](f: Function[V, U]): JavaPairRDD[K, U]
```

将 Key/ Value 型 RDD 中的每个元素的 Value 值，使用输入参数函数 f 进行变换，生成新的 RDD。

√ 示例：

mapValues 示例代码

```
1:  scala > val pairs = sc.parallelize(List ("apple","banana","berry",
"cherry","cumquat", "haw"),1).keyBy (_.length) //构建原始 RDD
    pairs: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[16]at keyBy
at <console>:12
2:  scala>pairs.mapValues(v => v + " " + v(0)).collect //生成将单词加单词首字
母的 RDD
    res0: Array[(Int, String)] = Array((5,apple a), (6,banana b), (5,berry
b), (6,cherry c), (7,cumquat c), (3,haw h))
```

示例代码的变换过程如图 4-20 所示，其原始 RDD 为单词长度与单词构成的 Key/ Value 型 RDD。第 2 行代码中指定的变换函数 v = > v + " " + v(0) 是取出 Value 值（即单词）及每个 Value 值第一个元素（即首字母），中间用空格连接，然后生成新的 RDD。

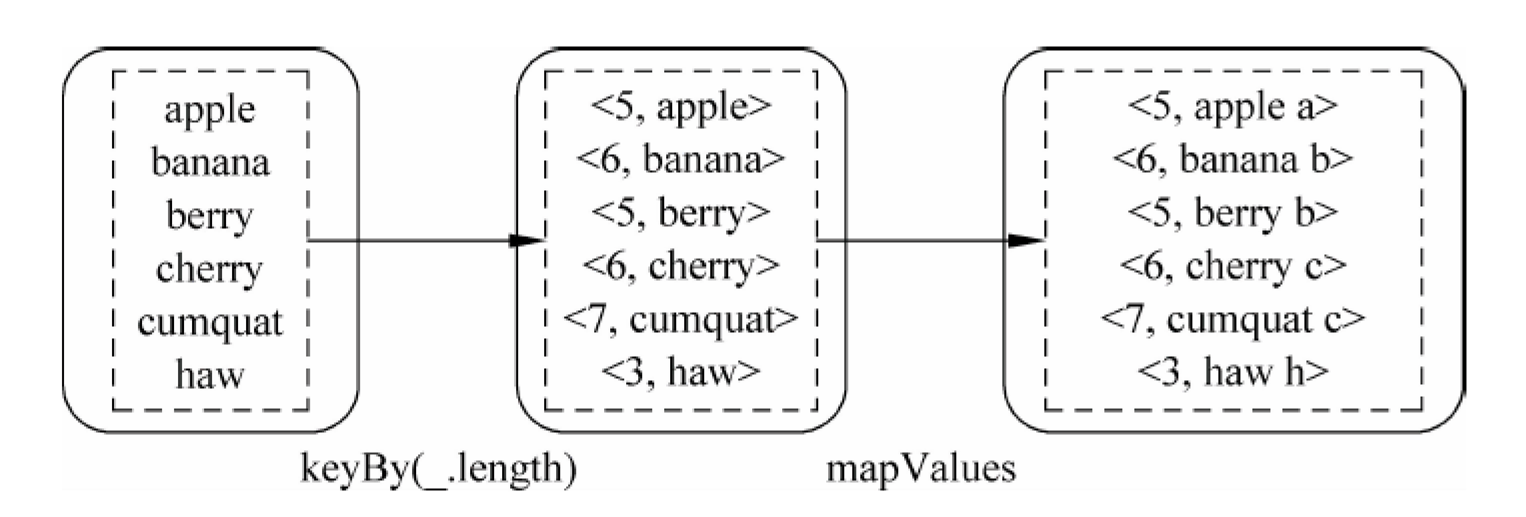


图 4-20 mapValues 算子变换过程

- **partitionBy——按 Key 值重新分区**

√ 算子函数格式：

```
-partitionBy(partitioner: Partitioner): JavaPairRDD[K, V]
```

将 Key/ Value 型 RDD 中的元素按照输入参数 partitionBy 指定的分区规则进行重新分区，生成新的 RDD。

√ 示例：

partitionBy 示例代码

```
1:  scala> val pairs = sc.parallelize(0 to 9,2).keyBy(x => x) //构建原始 RDD
    pairs: org.apache.spark.rdd.RDD[(Int, Int)] = MappedRDD[20]at keyBy at
<console>:12
2:  scala> pairs.glom.collect //显示原始 RDD 的分区情况
    res0: Array[Array[(Int, Int)]] = Array(Array((0,0), (1,1), (2,2), (3,3),
(4,4)), Array((5,5), (6,6), (7,7), (8,8), (9,9)))
3: scala> import org.apache.spark.HashPartitioner //导入 HashPartitioner
类库
    import org.apache.spark.HashPartitioner
4:  scala> val partitionedPairs = pairs.partitionBy(new HashPartitioner
(2)) //按照 Key 的 Hash 值进行重新分区
    partitionedPairs: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD
[21]at partitionBy at <console>:15
5:  scala> partitionedPairs.glom.collect //显示重新分区后的结果
    res1:  Array[Array[(Int, Int)]] = Array(Array((0,0), (2,2), (4,4), (6,
6), (8,8)), Array((1,1), (3,3), (5,5), (7,7),(9,9)))
```

示例代码的变换过程如图4-21所示，其原始RDD是由数字0~9构成的Key/Value型RDD，Key与Value为相同的数字，并且分为两个分区存放。然后使用partitionBy变换对这个RDD进行重新分区，分区的规则是计算每个元素Key的Hash值，Hash值相同的元素会放到同一个分区中，且仍然分为两个分区，最终构成新的RDD。

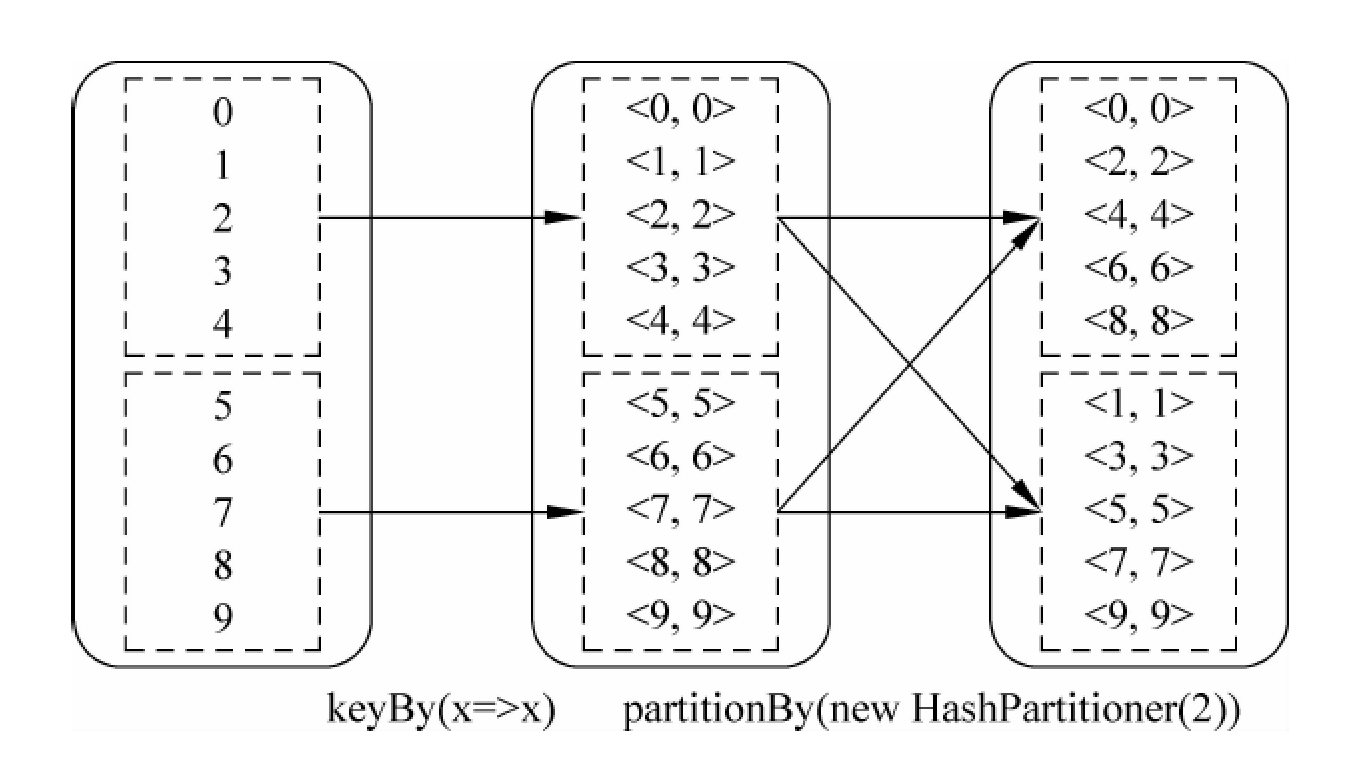


图4-21 partitionBy算子变换过程

- **reduceByKey——按Key值进行Reduce操作**

√ 算子函数格式：

```
-reduceByKey(func: Function2[V, V, V]): JavaPairRDD[K, V]
```

将 Key/ Value 型 RDD 中的元素按 Key 值进行 Reduce 操作，Key 值相同的 Value 值按照参数 func 的逻辑进行归并，然后生成新的 RDD。需要注意的是，reduceByKey 变换还有两个变型，分别为可以指定分区数量的 reduceByKey((func: Function2[V, V, V]), numPartitions: Int) 和可以指定分区类的 reduceByKey(partitioner: Partitioner, func: Function2[V, V, V])。

√ 示例：

reduceByKey 示例代码

```
1:  scala > val fruits = sc.parallelize(List("apple","banana","berry",
"cherry","cumquat", "haw"),1).keyBy(_.length)
    fruits: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD
[15]at keyBy at <console>:22
2:  scala> fruits.reduceByKey(_ + " " + _).collect
    res6: Array[(Int, String)] = Array((6,banana cherry), (7,cumquat), (3,
haw), (5,apple berry))
```

示例代码的变换过程如图 4-22 所示。输入 RDD 为以单词长度为 Key，单词为 Value 的 Key/ Value 型 RDD。reduceByKey 变换中指定的函数为将长度相同的单词用空格进行连接。

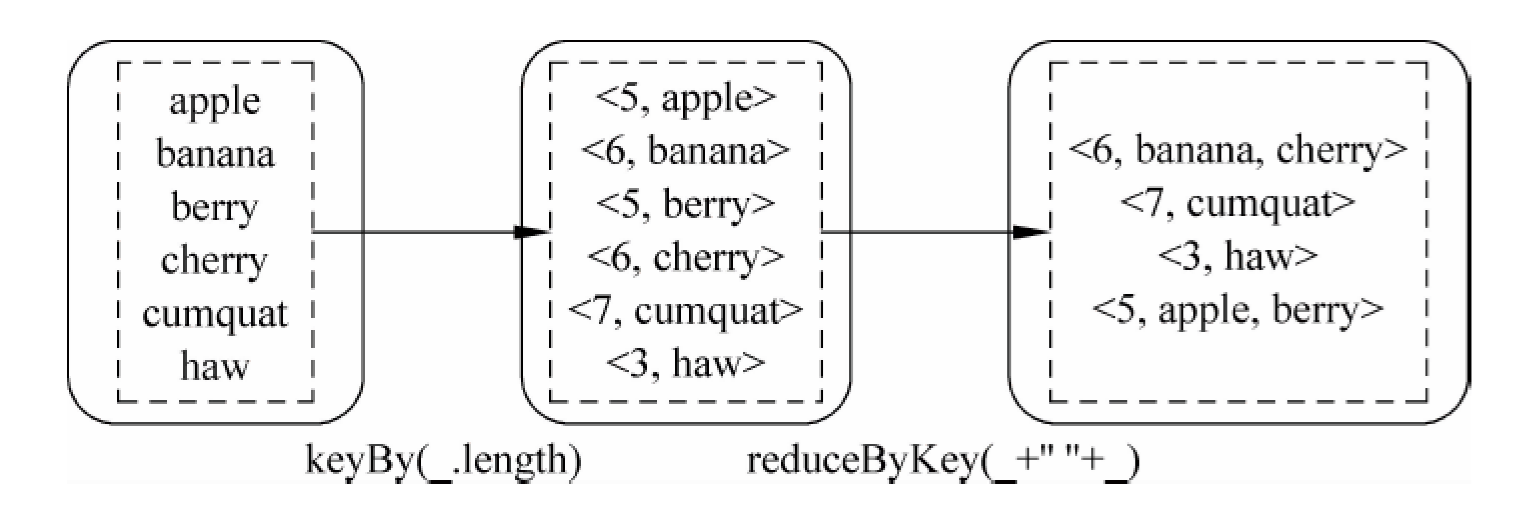


图 4-22 reduceByKey 算子变换过程

- **sortByKey——按 Key 值排序**

√ 算子函数格式：

```
-sortByKey(ascending: Boolean, numPartitions: Int): JavaPairRDD[K, V]
```

对原 RDD 中的元素按照 Key 值进行排序，并可使用参数 ascending 指定按照升序或者降序进行排列，排序后的结果生成新的 RDD，新 RDD 的分区数量可以由参数 numPartitions 指定，默认采用与原 RDD 相同的分区数。

√ 示例：

sortByKey 示例代码

```
1:  scala > val words = sc.parallelize(List("apple","banana","cat","door"),
1).keyBy(_.length)//构建原始 RDD
    words: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18]at
parallelize at <console>:12
2:  scala > words.sortByKey(true).collect //按单词长度排序
    res0: Array[(Int, String)] = Array((3,cat), (4,door), (5,apple), (6,
banana))
```

示例代码的变换过程如图 4-23 所示，输入 RDD 为单词长度（Key）和单词（Value）构成的 Key/ Value 型 RDD。使用 sortByKey 变换，对 Key 值进行升序排序，即将各个元素按单词长度进行升序的排列，生成新的 RDD。

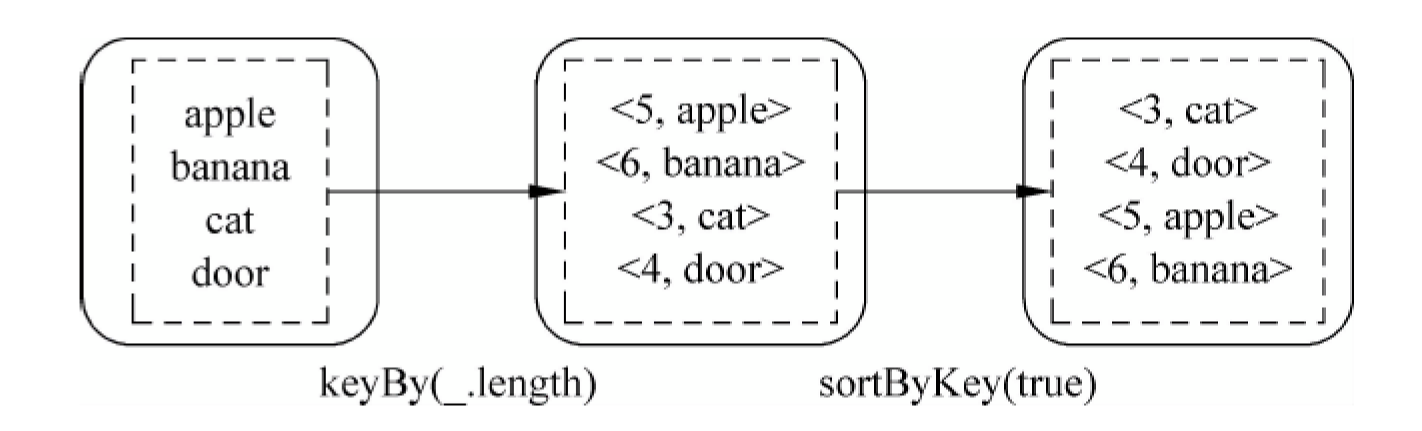


图 4-23 sortByKey 算子变换过程

- **values**

√ 算子函数格式：

```
- values(): JavaRDD[V]
```

对 Key/ Value 型的 RDD 进行取值操作，即将 RDD 转化为只有元素的 Value 值构成的新 RDD。

√ 示例：

values 示例代码

```
1:  scala > val words = sc.parallelize(List("apple","banana","cat","door"),
    1).keyBy(_.length) //构建原始 RDD
    words: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18]at
parallelize at <console>:12
2:  scala > words.values.collect //取出 Value 值
    res0: Array[String] = Array(apple, banana, cat, door)
```

示例代码的变换过程如图 4-24 所示，输入 RDD 为单词长度(Key)和单词(Value)构成的 Key/ Value 型的 RDD。对其使用 values 变换，取出了原 RDD 中所有 Value 值，即单词，生成新的 RDD。

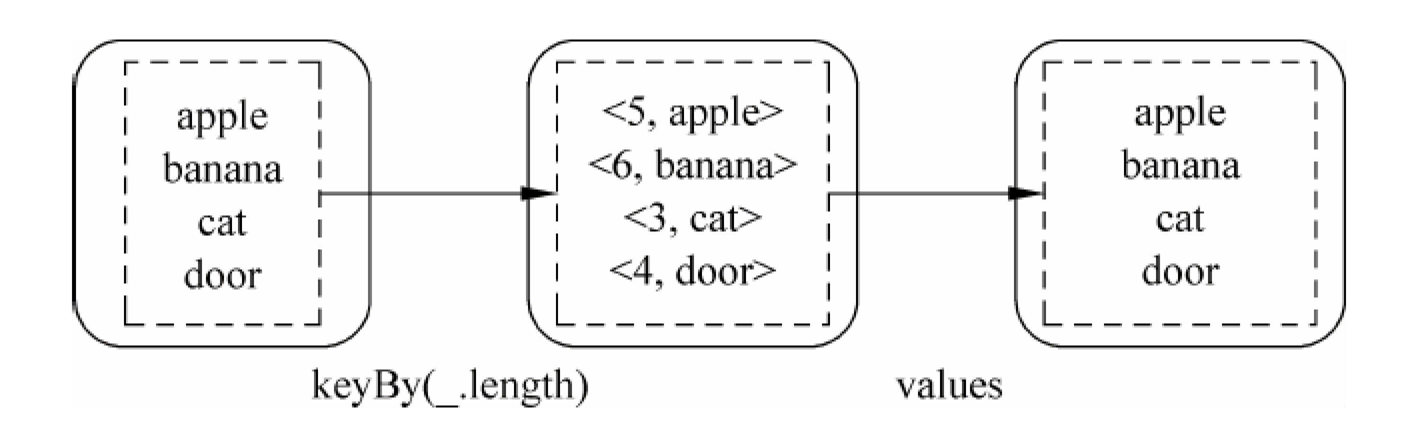


图 4-24　values 算子变换过程

4.2.2.3　对两个 Key-Value 型 RDD 进行变换

- **cogroup——按 Key 值聚合**

√ 算子函数格式：

```
- cogroup[W](other: JavaPairRDD[K, W], numPartitions: Int): JavaPairRDD
[K, (Iterable[V], Iterable[W])]
```

对于任何一个在 Key/ Value 型的原始 RDD 或在 Key/ Value 型的输入 RDD(输入参数 other)中存在的 Key，cogroup 变换会寻找在两个 RDD 中相同 Key 值的元素，将所有这些元素的 Value 聚合构成一个序列，然后与 Key 值生成新的 RDD。与后面会讲解的 join 变换不同之处在于，在新 RDD 中出现的 Key 值并不要求在两个 RDD 中都存在。

√ 示例：

cogroup 示例代码

```
1:  scala> val pair1 = sc.parallelize(List((1,"a"),(2,"b"),(3,"c")),2) //构
建原始 RDD
    pair1:  org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollection
RDD[0]at parallelize at <console>:21
2:  scala> val pair2 = sc.parallelize(List((1,"apple"),(2,"banana")),2) //
构建输入参数 RDD
    pair2: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollection
RDD[1]at parallelize at <console>:21
3:  scala> val cogrouped = pair1.cogroup(pair2,1).collect //按 Key 值进行
联结
    cogrouped: Array[(Int, (Iterable[String], Iterable[String]))] = Array
((1,(CompactBuffer(a), CompactBuffer
    (apple))), (3,(CompactBuffer(c),CompactBuffer())), (2,(CompactBuffer
(b), CompactBuffer(banana))))
```

示例代码的变换过程如图 4-25 所示，原始 RDD 为序号为 Key、字母为 Value 的 Key/ Value 型 RDD，输入参数 RDD 为序号为 Key、单词为 Value 的 Key/ Value 型 RDD。经过 cogroup 变换后，所有出现过的序号，以及相应的字母和单词汇聚在一起构成列表，从而得到新的 RDD。我们可以看到，虽然序号为 3、字母为 c 的元素只在原始 RDD 中出现了，也仍会汇聚出一个没有单词的元素在新 RDD 中出现。

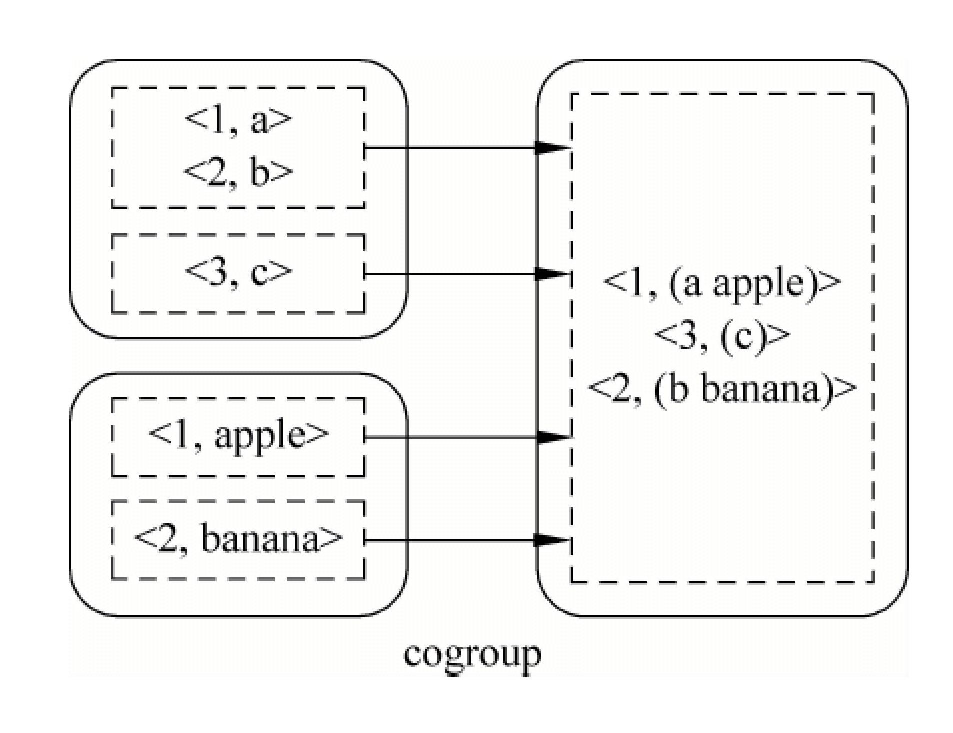


图 4-25 cogroup 算子变换过程

- **join——按 Key 值联结**

√ 算子函数格式：

```
-join[W](other: JavaPairRDD[K, W]): JavaPairRDD[K, (V, W)]
```

对于在 Key/ Value 型的原始 RDD 和在 Key/ Value 型的输入 RDD（输入参数 other）中同时存在的 Key，join 变换会寻找在两个 RDD 中相同 Key 值的元素，将所有这些元素的 Value 联结构成一个序列，然后与 Key 值生成新的 RDD。与前面讲解的 cogroup 变换不同之处在于，在新 RDD 中出现的 Key 值要求在两个 RDD 中都存在。

√ 示例：

join 示例代码

```
1:  scala> val pair1 = sc.parallelize(List((1,"a"),(2,"b"),(3,"c")),2) //构建原始 RDD
    pair1: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[0]at parallelize at <console>:21
2:  scala> val pair2 = sc.parallelize(List((1,"apple"),(2,"banana")),2) //构建输入参数 RDD
    pair2: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[1]at parallelize at <console>:21
3:  scala> pair1.join(pair2,1).collect //按 Key 值进行联结
    res0: Array[(Int, (String, String))] = Array((2, (b,banana)), (1, (a,apple)))
```

示例代码的变换过程如图 4-26 所示，原始 RDD 为序号为 Key、字母为 Value 的 Key/ Value 型 RDD，输入参数 RDD 为序号为 Key、单词为 Value 的 Key/ Value 型 RDD。经过 join 变换后，在两个 RDD 中都出现过的序号，以及相应的字母和单词汇聚在一起构成列表，从而得到新的 RDD。我们可以看到，序号为 3、字母为 c 的元素由于在输入参数 RDD 中没有出现过，因此不会出现在新 RDD 中。

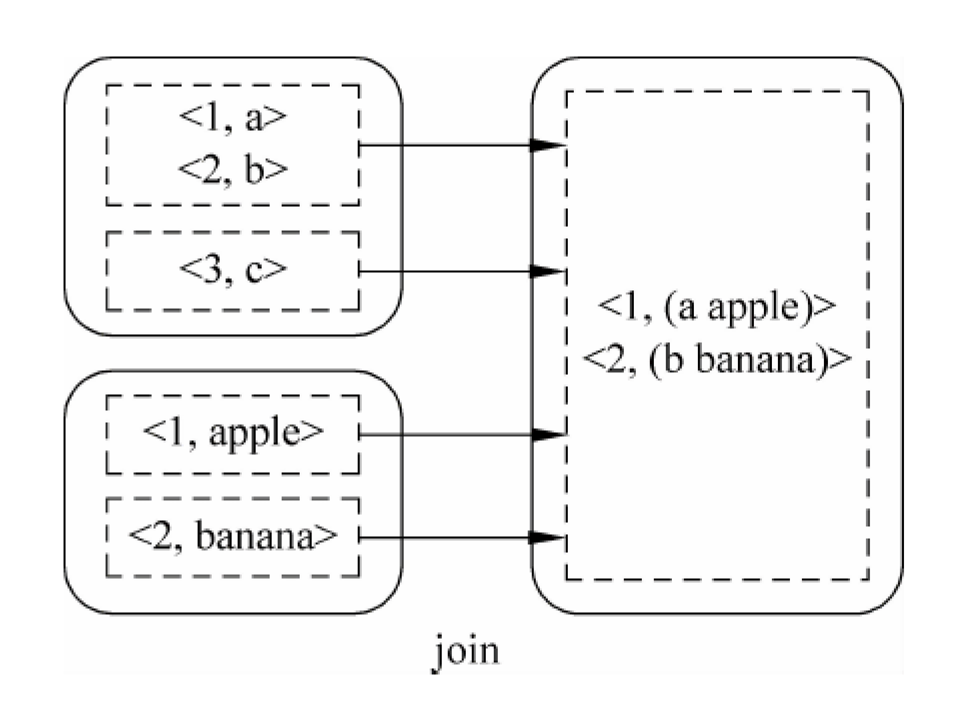


图 4-26　join 算子变换过程

- **leftOuterJoin——按 Key 值进行左外联结**

√ 算子函数格式：

```
- leftOuterJoin[W](other: JavaPairRDD[K, W]): JavaPairRDD[K, (V, Optional[W])]
```

对两个 RDD 进行左连接。

√ 示例：

leftOuterJoin 示例代码

```
1:  scala> val a = sc.parallelize(List("a","boy","cafe"),1).keyBy(_.length)
    a: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[3]at keyBy at <console>:12
2:  scala>a.collect
    res0: Array[(Int, String)] = Array((1,a), (3,boy), (4,cafe))
3: scala> val b = sc.parallelize(List("dog","enjoy","fate"),1).keyBy(_.length)
    b: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[5]at keyBy at <console>:12
4:  scala>b.collect
    res1: Array[(Int, String)] = Array((3,dog), (5,enjoy), (4,fate))
5:  scala>a.leftOuterJoin(b).collect
    res2: Array[(Int, (String, Option[String]))] = Array((4,(cafe,Some(fate))), (1,(a,None)), (3,(boy,Some(dog))))
```

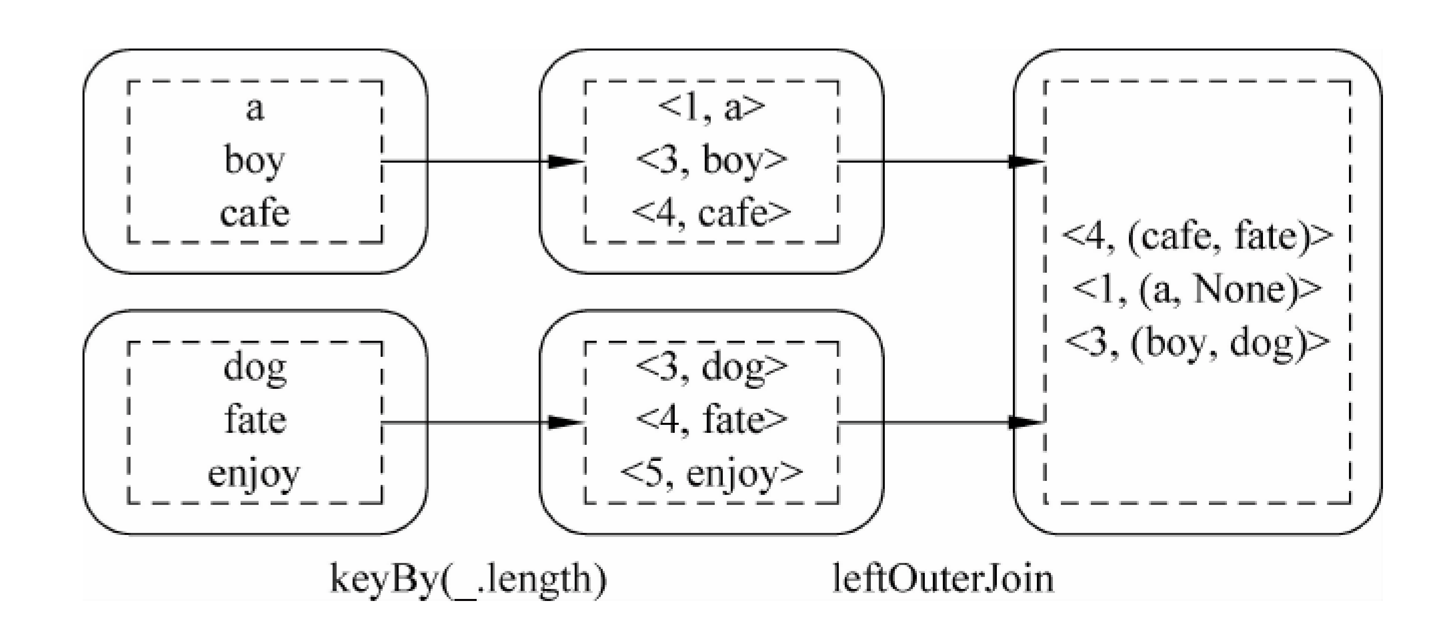


图 4-27　leftOuterJoin 算子变换过程

- **rightOuterJoin——按 Key 值进行右外联结**

√ 算子函数格式：

```
-rightOuterJoin[W](other: JavaPairRDD[K, W]): JavaPairRDD[K, (Optional
[V], W)]
```

对两个 RDD 进行右连接。

√ 示例：

rightOuterJoin 示例代码

```
1:  scala> val a = sc.parallelize(List("a","boy","cafe"),1).keyBy(_.
length)
    a: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[3]at keyBy at
<console>:12
2:  scala>a.collect
    res0: Array[(Int, String)] = Array((1,a), (3,boy), (4,cafe))
3:  scala> val b = sc.parallelize(List("dog","enjoy","fate"),1).keyBy(_.
length)
    b: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[5]at keyBy at
<console>:12
4:  scala>b.collect
    res1: Array[(Int, String)] = Array((3,dog), (5,enjoy), (4,fate))
5:  scala>a.rightOuterJoin(b).collect
    res2: Array[(Int, (Option[String], String))] = Array((4,(Some(cafe),
fate)), (3,(Some(boy),dog)), (5,(None,enjoy)))
```

- **subtractByKey——按 Key 值求补**

√ 算子函数格式：

```
-subtractByKey[W](other: RDD[(K, W)])
```

对于只在 Key/ Value 型的原始 RDD 中出现，而不在 Key/ Value 型的输入 RDD

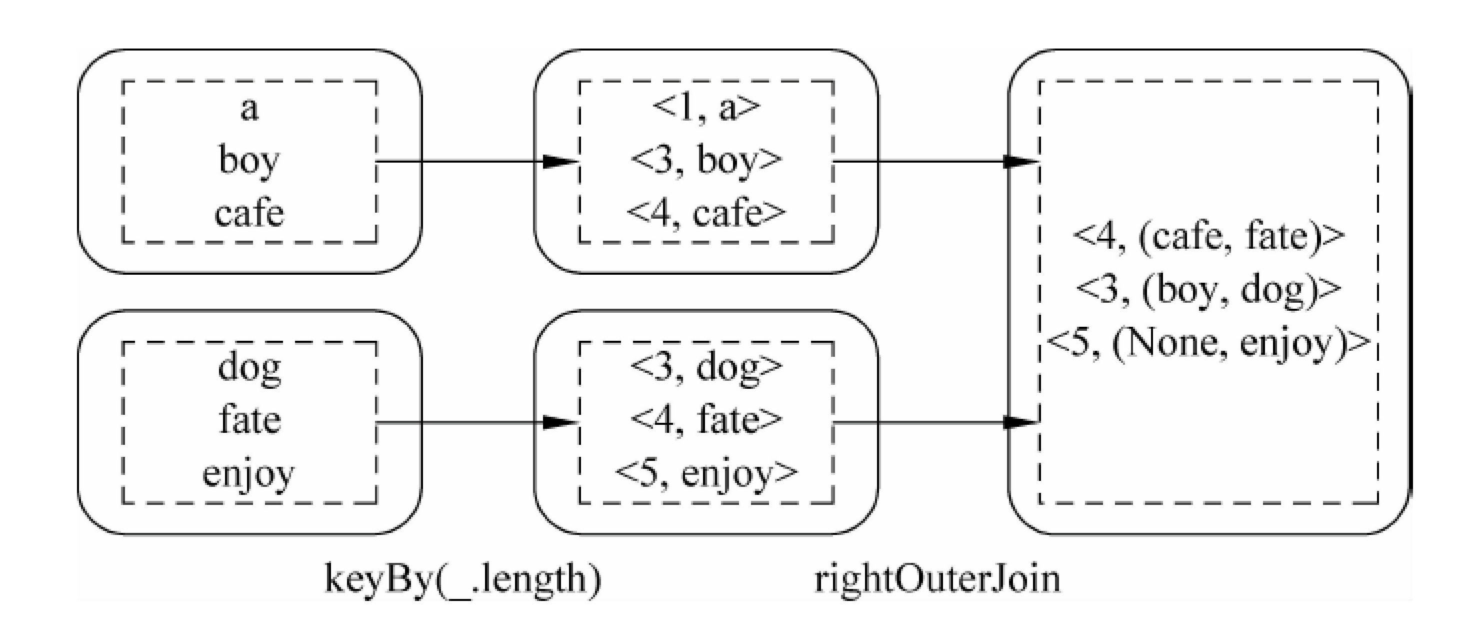


图 4-28 rightOuterJoin 算子变换过程

（输入参数 other）中出现的 Key 值，将所有这些 Key 值对应的 Key/ Value 对元素，生成新的 RDD。

√ 示例：

subtractByKey 示例代码

```
1： scala > val words = sc.parallelize(List("apple","banana","cat","door"),
1).keyBy(_.length) //构建原始 RDD
    words: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18]at
parallelize at <console>:12
2： scala > val others = sc.parallelize(List("boy","girl"),1).keyBy(_.
length) //构建输入参数 RDD
    others: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18]at
parallelize at <console>:12
3： scala > words.subtractByKey(others).collect //按 Key 值求补
    res0: Array[(Int, String)] = Array((5,apple), (6,banana))
```

示例代码的变换过程如图 4-29 所示，原始 RDD 和输入参数 RDD 均为单词字母数为 Key、单词为 Value 的 Key/ Value 型 RDD。经过 subtractByKey 变换后，生成的 RDD 中只包含 Key 值仅出现在原始 RDD 中（Key 值 5 和 6）的 Key/ Value 对。

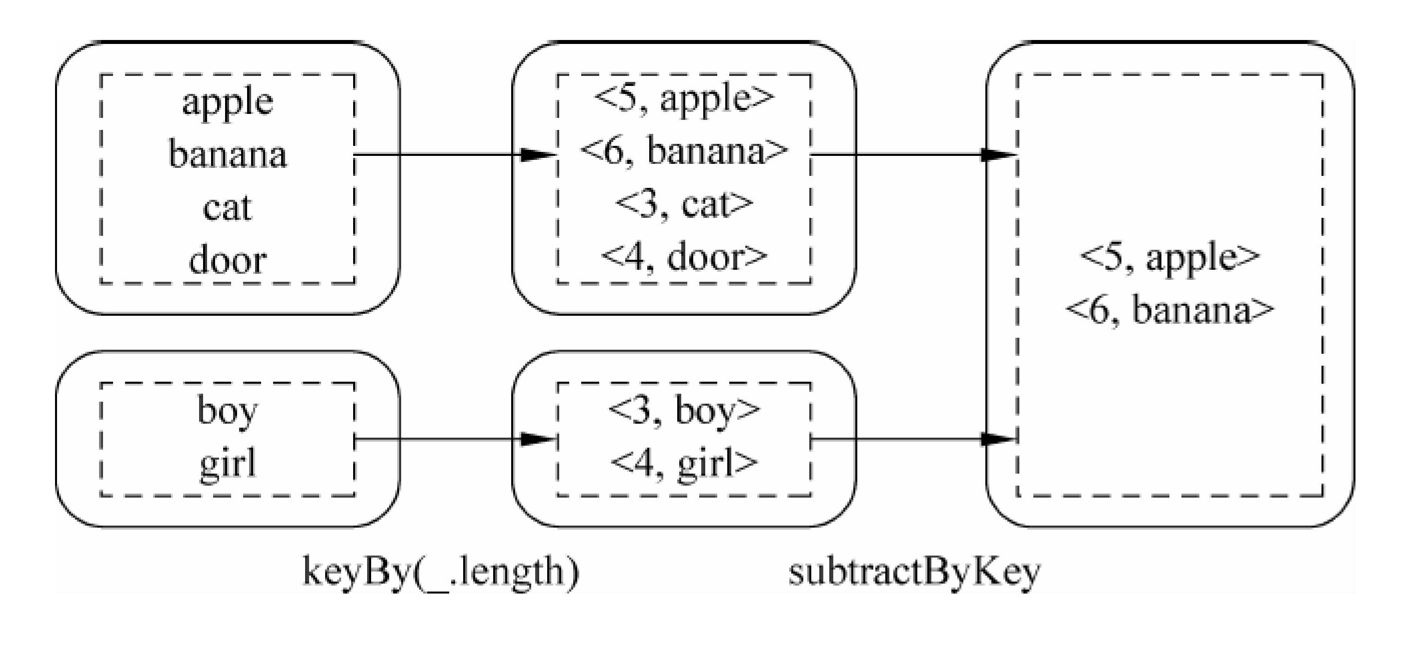


图 4-29 subtractByKey 算子变换过程

4.3 行动算子

在前面介绍 Spark 原理时,我们已经了解到 Spark 中的“惰性执行”机制,也就是 Spark 中的变换算子并不是在运行到相应语句时就会立即执行,而是在遇到本节将要介绍的行动(Action)算子语句时,才会真正触发 Spark 的任务调度开始进行计算,这也是为什么这类算子被称为“行动”算子的原因。在 Spark 中,行动算子的数量也相对较多。因此,为了理解方便,我们根据这些行动算子的输出结果,将它们分为两类:①数据运算类,该类算子的作用是触发 RDD 计算,并得到计算结果返回给 Spark 程序或 Shell 界面;②数据存储类,该类算子在触发 RDD 计算后,将结果保存到外部存储系统中,例如 HDFS 文件系统或数据库。下面我们就对这两类算子分别进行介绍。

4.3.1 数据运算类行动算子

在介绍这类算子时,我们从大家在 Hadoop 中可能已经比较熟悉的 reduce 算子开始以帮助理解,然后按照算子的字母顺序进行介绍。

- **reduce——Reduce 操作**

√ 算子函数格式:

```
-reduce(f:Function2[T, T, T]):T
```

对 RDD 中的每个元素依次使用指定的函数 f 进行运算,并输出最终的计算结果。需要注意的是,Spark 中的 reduce 操作与 Hadoop 中的 reduce 操作并不一样。在 Hadoop 中,reduce 操作是将指定的函数作用在 Key 值相同的全部元素上。而 Spark 的 reduce 操作则是对所有元素依次进行相同的函数计算。

√ 示例:

reduce 示例代码

```
1:  scala> val nums = sc.parallelize(0 to 9,5)//构建由数字 0 -9 构成的 RDD
    nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at
parallelize at <console>:12
2:  scala> nums.reduce(_ + _) //计算 RDD 中所有数字的和
    res0: Int = 45
```

- **aggregate——聚合操作**

√ 算子函数格式：

```
- aggregate[U](zeroValue: U)(seqOp: Function2[U, T, U], combOp: Function2
[U, U, U]): U
```

aggregate 操作使用参数 seqOp 指定的函数对每个分区里面的元素进行聚合，然后用参数 combOp 指定的函数将每个分区的聚合结果进行再次聚合，在进行 combOp 聚合时，计算的初始值由参数 zeroValue 指定。需要注意的是，聚合操作在进行每个分区的 seqOp 操作以及最终的 combOp 操作时，生成结果的数据类型可能与 RDD 中的元素不一样。

√ 示例：

aggregate 示例代码

```
1:  scala> val rdd = sc.parallelize(List(1,2,3,4,5,6), 2) //构建原始 RDD
     rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
parallelize at <console>:12
2:  scala> rdd.glom.collect //显示原始 RDD
    res0: Array[Array[Int]] = Array(Array(1, 2, 3), Array(4, 5, 6))
3:  scala> rdd.aggregate(0)(_+_, Math.max(_,_)) //对原始 RDD 进行聚合，求每个
分区元素相加后的最大值
    res1: Int = 15
```

示例代码的执行过程如图 4-30 所示。原始 RDD 为数字 1 ~6 的序列，分别存放在两个分区中。第 3 行代码的 aggregate 操作指定的 seqOp 函数是_ + _，及每个分区的所有元素依次相加。指定的 combOp 函数为 Math. max(_,_)，即求每个分区的元素相加后的和构成的数据集中的最大值。最后的计算结果是第 2 个分区 4、5、6 相加后得到的 15。

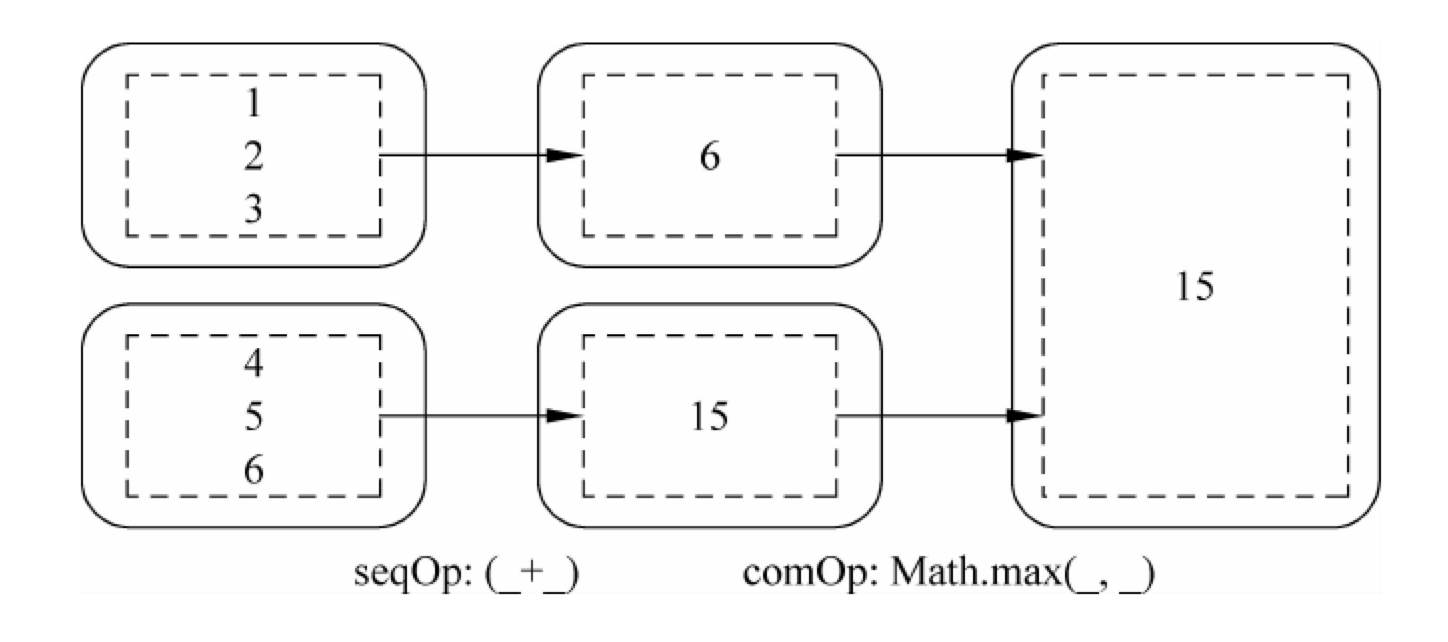


图 4-30 aggregate 算子变换过程

- **collect——收集元素**

√ 算子函数格式：

```
-collect(): List[T]
```

collect 的作用是以数组格式返回 RDD 内的所有元素。

√ 示例：

collect 示例代码

```
1:  scala>val data=sc.parallelize(List(1,2,3,4,5,6,7,8,9,0),2) //构建原始 RDD
    data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:12
2:  scala>data.collect //显示原始 RDD 中的元素
    res0:Array[Int]=Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
```

- **collectAsMap——收集 Key/ Value 型 RDD 中的元素**

√ 算子函数格式：

```
-collectAsMap(): Map[K, V]
```

收集 Key/ Value 型 RDD 中的元素，并以 Map 数据类型的方式返回结果。

√ 示例：

collectAsMap 示例代码

```
1:  scala>val pairRDD=sc.parallelize(List((1,"a"),(2,"b"),(3,"c"),(4,"d")),2) //构建原始 RDD
    pairRDD: org.apache.spark.rdd.RDD[(Int, String)]=ParallelCollectionRDD[9]at parallelize at <console>:12
2:  scala>pairRDD.collectAsMap //以 Map 的方式返回 RDD 中的结果
    res0: scala.collection.Map[Int,String]=Map(2->b, 4->d, 1->a, 3->c)
```

- **count——计算元素个数**

√ 算子函数格式：

```
-count(): Long
```

计算并返回 RDD 中元素的个数。

√ 示例：

count 示例代码

```
scala> val rdd = sc.parallelize(List(1,2,3,4,5,6),2)//构建原始 RDD
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:12
scala> rdd.count //计算 RDD 中元素的个数
res0: Long = 6
```

- **countByKey——按 Key 值统计 Key/ Value 型 RDD 中的元素个数**

√ 算子函数格式：

```
- countByKey(): Map[K, Long]
```

计算 Key/ Value 型 RDD 中每个 Key 值对应的元素个数，并以 Map 数据类型返回统计结果。

√ 示例：

countByKey 示例代码

```
1: scala> val pairRDD = sc.parallelize(List(("fruit","Apple"),("fruit","Banana"),("fruit","Cherry"),("vegetable","bean"),("vegetable","cucumber"),("vegetable","pepper")),2)//构建原始 RDD
   pairRDD: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[3]at parallelize at <console>:12
2: scala> pairRDD.countByKey //统计原始 RDD 中每个物品类型下的物品数量
   res0: scala.collection.Map[String,Long] = Map(fruit -> 3, vegetable -> 3)
```

示例代码的第 1 行构建了一个 Key/ Value 类型的 RDD，其中 Key 为物品的类型名称，例如水果（fruit），Value 为物品名称，例如苹果（Apple）。第 2 行代码的目的是统计每个类型下的物品数量。

- **countByValue——统计 RDD 中元素值出现的次数**

√ 算子函数格式：

```
- countByValue(): Map[T, Long]
```

计算 RDD 中每个元素的值出现的次数，并以 Map 数据类型返回统计结果。

√ 示例：

countByValue 示例代码

```
1:  scala > val num = sc.parallelize(List(1,1,1,2,2,3),2) //构建原始 RDD
     num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at
parallelize at <console>:12
2:  scala > num.countByValue//统计原始 RDD 中每个数字出现的次数
    res0: scala.collection.Map[Int,Long] = Map(2 -> 2, 1 -> 3, 3 -> 1)
```

示例代码的第 1 行构建了一个由数字构成的 RDD，第 2 行代码的目的是统计每个数字出现的次数。

- **first——得到首个元素**

√ 算子函数格式：

-first(): T

返回 RDD 中的第一个元素。

√ 示例：

first 示例代码

```
1:  scala > val words = sc.parallelize(List("first","second","third"),1)//构
建原始 RDD
    words: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[5]at
parallelize at <console>:12
2:  scala > words.first //得到首个元素
    res0: String = first
```

- **glom——返回分区情况**

√ 算子函数格式：

-glom():JavaRDD[List[T]]

原始 RDD 每个分区中的元素放到一个序列中，并汇集所有分区构成的序列生成新的 RDD 返回。

√ 示例：

glom 示例代码

```
1:  scala > val num = sc.parallelize(0 to 10,4) //构建原始 RDD
```

```
    num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at
parallelize at <console>:12
2:  scala> num.glom.collect //显示分区中的元素分布情况
    res0: Array[Array[Int]] = Array(Array(0, 1), Array(2, 3, 4), Array(5, 6,
7), Array(8, 9, 10))
```

- **fold——合并**

√ 算子函数格式:

```
- fold(zeroValue: T)(f: Function2[T, T, T]): T
```

将 RDD 中每个分区中的元素使用指定的函数参数 f 做合并计算,然后再加所有分区生成的结果使用 f 函数做合并。在分区内和不同分区间进行合并的初始值由 zeroValue 指定。

√ 示例:

fold 示例代码

```
1:  scala> val words = sc.parallelize(List("A","B","C","D"),2) //构建原始
RDD
    words: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2]at
parallelize at <console>:12
2: scala> words.glom.collect //显示原始 RDD 的分区情况
    res0: Array[Array[String]] = Array(Array(A, B), Array(C, D))
3: scala> words.fold("|")(_ + "." + _) //对原始 RDD 执行合并
    res1:  String = |.|.A.B.|.C.D
```

示例代码的执行过程如图 4-31 所示。示例代码第 1 行构建了一个由字母序列构成的 RDD,存放在 2 个分区中。第 3 行的 fold 操作指定的 f 函数为_ + "." + _,即将两个字符串用.连接起来。我们看到最终的结果为|.|.C.D.|.A.B。在结果的最前面有两个|.字符,是因为在分区内合并时要由初始值带入一个|.,在所有分区结果合并时,又要带入一个|.。

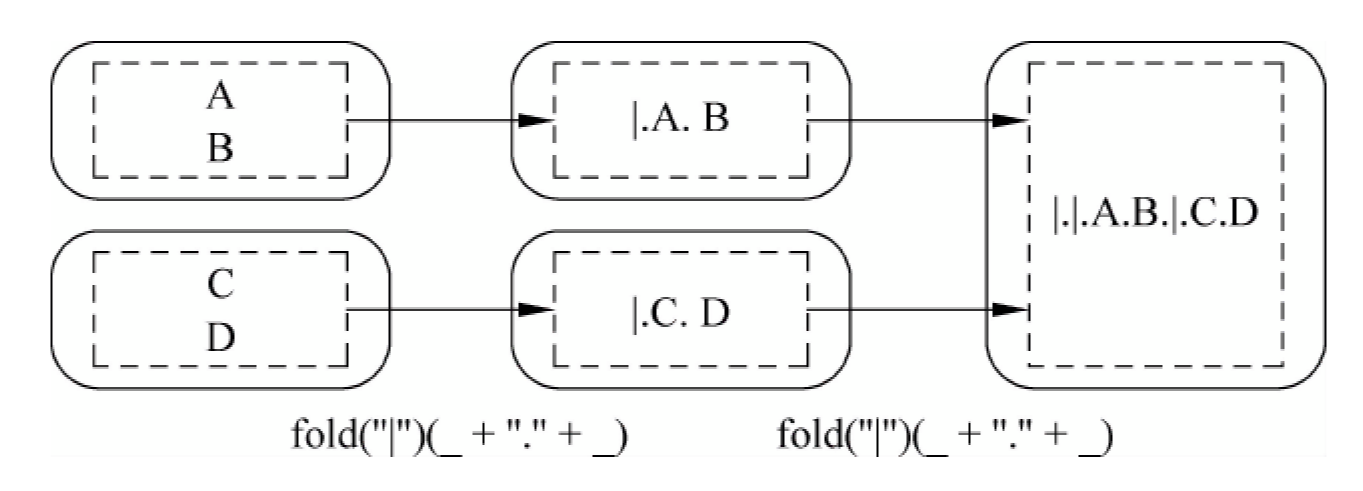


图 4-31　fold 算子变换过程

- **foreach——逐个处理 RDD 元素**

√ 算子函数格式:

```
- foreach(f: VoidFunction[(K, V)]): Unit
```

对 RDD 中的每个元素,使用参数 f 指定的函数进行处理。

√ 示例:

foreach 示例代码

```
1:  scala > val words = sc.parallelize(List("A","B","C","D"),2)//构建原始 RDD
    words: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[9]at
parallelize at <console>:21
2:  scala > words.foreach(x => println(x + " is a letter.")) //打印输出每个单
词构造的一句话
    C is a letter.
    A is a letter.
    D is a letter.
    B is a letter.
```

- **lookup——查找元素**

√ 算子函数格式:

```
- lookup(key: K): List[V]
```

在 Key/ Value 型的 RDD 中,查找与参数 key 相同 Key 值的元素,并得到这些元素的 Value 值构成的序列。

√ 示例:

lookup 示例代码

```
1:  scala > val pairs = sc.parallelize(List("apple","banana","berry","
cherry","cumquat","haw"),1).keyBy (_.length) //构建原始 RDD
    pairs: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[13]
at keyBy at <console>:21
2:  scala > pairs.collect
    res18: Array[(Int, String)] = Array((5,apple), (6,banana), (5,berry),
(6,cherry), (7,cumquat), (3,haw))
3:  scala > pairs.lookup(5) //查找长度为 5 的单词
    res19: Seq[String] = WrappedArray(apple, berry)
```

示例代码的第 1 行是构建一个 Key/ Value 型的 RDD,Key 为单词长度,Value 为单

词。第 3 行代码的目标是查找长度为 5 的单词。

- **max——求最大值**

√ 算子函数格式：

```
-max(comp: Comparator[(K, V)]): (K, V)
```

返回 RDD 中值最大的元素，可以通过参数 comp 指定元素间比较大小的方法。

√ 示例：

max 示例代码

```
1:  scala> val nums = sc.parallelize(0 to 9,1) //构建由 0-9 组成的 RDD
nums: org. apache. spark. rdd. RDD [Int ] = ParallelCollectionRDD [18 ] at
parallelize at <console>:12
2:  scala> nums.max //寻找 RDD 中最大的值
res0: Int = 9
```

- **min——求最小值**

√ 算子函数格式：

```
-min(comp: Comparator[T]): T
```

返回 RDD 中值最小的元素，可以通过参数 comp 指定元素间比较大小的方法。

√ 示例：

min 示例代码

```
1:  scala> val nums = sc.parallelize(0 to 9,1) //构建由 0-9 组成的 RDD
    nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18]at
parallelize at <console>:12
2:  scala> nums.min //寻找 RDD 中最小的值
    res0: Int = 0
```

- **take——获取前 n 个元素**

√ 算子函数格式：

```
-take(num: Int): List[T]
```

以数组的方式返回 RDD 中的前 num 个元素。

√ 示例：

take 示例代码

```
1:  scala > val sample = sc.parallelize(0 to 4,1) //构建 0 -4 构成的 RDD
    sample: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0]at
parallelize at <console>:15
2:  scala > sample.take(2) //返回 RDD 中的前两个元素
    res0: Array[Int] = Array(0, 1)
```

- **takeOrdered——获取排序后的前 n 个元素**

√ 算子函数格式：

```
-takeOrdered(num: Int): List[T]
```

以数组的方式返回 RDD 中的元素经过排序后的前 num 个元素。

√ 示例：

takeOrdered 示例代码

```
1:  scala > val sample = sc.parallelize(List(2,1,4,3,0),1) //构建 0 -4 乱序后
    构成的 RDD
    sample: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1]at
parallelize at <console>:15
2:  scala > sample.takeOrdered(2) //返回 RDD 排序后的前 2 个元素
    res0: Array[Int] = Array(0, 1)
```

- **takeSample——提取 n 个样本**

√ 算子函数格式：

```
-takeSample(withReplacement: Boolean, num: Int, seed: Long): List[T]
```

随机提取 RDD 中一定数量的样本元素，并以数组方式返回。参数 withReplacement 指定是否为放回取样，参数 num 指定提取的样本数量，参数 seed 为随机种子。

√ 示例：

takeSample 示例代码

```
1:  scala > val sample = sc.parallelize(0 to 9,1) //构建 0 -9 构成的 RDD
    sample: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0]at
parallelize at <console>:12
```

```
2:  scala > sample.takeSample(true,3,5) //以有放回取样的方式选取3 个样本元素
    res0: Array[Int] = Array(8, 2, 8)
3:  scala > sample.takeSample(false,3,5) //以无放回取样的方式选取 3 个样本元素
    res1: Array[Int] = Array(4, 3, 5)
```

- **top——寻找值最大的前几个元素**

√ 算子函数格式：

```
-top(num: Int, comp: Comparator[T]): List[T]
```

返回 RDD 中值最大的前 num 个元素，可以通过参数 comp 指定元素间比较大小的方法。

√ 示例：

top 示例代码

```
1:  scala > val nums = sc.parallelize(0 to 9,1) //构建由 0 - 9 组成的 RDD
    nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at parallelize at <console>:12
2:  scala > nums.top(3) //寻找 RDD 中最大的三个元素
    res0: Array[Int] = Array(9, 8, 7)
```

4.3.2 存储型行动算子

- **saveAsObjectFile——存储为二进制文件**

√ 算子函数格式：

```
-saveAsObjectFile(path: String): Unit
```

将 RDD 转换为序列号对象后，以 Hadoop SequenceFile 文件格式保存，保存路径由参数 path 指定。

√ 示例：

saveAsObjectFile 示例代码

```
1:  scala > val data = sc.parallelize(0 to 9,1) //构建 0 - 9 组成的 RDD
    data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[40] at parallelize at <console>:12
2:  scala > data.saveAsObjectFile("obj")//将 RDD 以 SequenceFile 文件格式保存,文件名为 obj
```

- **saveAsTextFile——存储为文本文件**

√ 算子函数格式：

```
-saveAsTextFile(path: String): Unit
```

将 RDD 以文本文件格式保存，保存路径由参数 path 指定。如果要节省存储空间，还可以选择使用该接口的另一种方式 saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec])，可以使用参数 codec 指定压缩方式。

√ 示例：

saveAsTextFile 示例代码

```
1:  scala> val data = sc.parallelize(0 to 9,1) //构建 0 - 9 组成的 RDD
    data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[40] at
parallelize at <console>:12
2:  scala> data.saveAsObjectFile("text") //将 RDD 以文本文件格式保存,文件名
为 text
```

- **saveAsNewAPIHadoopFile——存储为 Hadoop 文件**

√ 算子函数格式：

```
-saveAsNewAPIHadoopFile[F <: OutputFormat[_, _]](path: String, keyClass:
Class[_], valueClass: Class[_], outputFormatClass: Class[F], conf:
Configuration): Unit
```

将 Key/ Value 型 RDD 存储成 Hadoop 文件，输出格式由参数 F 指定，保存路径由参数 path 指定。

√ 示例：

saveAsNewAPIHadoopFile 示例代码

```
1:  scala> val pairs = sc.parallelize(List("apple","banana","berry","
cherry","cumquat","haw"),1).keyBy(_.length) //构建原始 RDD
    pairs: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[13]
at keyBy at <console>:21
2:  scala> pairs.saveAsNewAPIHadoopFile[TextOutputFormat[LongWritable,
String]](hFile1)
```

- **saveAsNewAPIHadoopDataset——存储为 Hadoop 数据集**

√ 算子函数格式：

```
-saveAsNewAPIHadoopDataset(conf: Configuration): Unit
```

将 Key/ Value 型 RDD 保存为 Hadoop 数据集，该接口一般用于将 RDD 存储到如 HBase 之类的数据库中。

√ 示例：

saveAsNewAPIHadoopDataset 示例代码

```
Configuration conf = HBaseConfiguration.create() //构造 HBase 配置
conf.set(TableInputFormat.INPUT_TABLE, "user") //设置使用的 HBase 表名
pairRDD.saveAsNewAPIHadoopDataset(conf) //将 RDD 数据写入 HBase 中
```

4.4 缓存算子

为了提高计算效率，Spark 采用了两个重要机制：①基于分布式内存数据集进行运算，也就是我们已经熟知的 RDD；②变换算子的惰性执行（Lazy Evaluation），即 RDD 的变换操作并不是在运行到该行代码时立即执行，而仅记录下转换操作的操作对象。只有当运行到一个行动算子代码时，变换操作的计算逻辑才真正执行。这两个机制帮助 Spark 提高了运算效率，但正如"硬币都有两面"一样，在带来提升性能的好处的同时，这两个机制也留下了隐患。例如：①如果在计算过程中，需要反复使用某个 RDD，而该 RDD 需要经过多次变换才能得到，则每次使用该 RDD 时都需要重复这些变换操作，这种运算效率是很低的；②在计算过程中数据存放在内存中，如果出现参与计算的某个节点出现问题，则存放在该节点内存中的 RDD 数据会发生损坏。如果损坏的也是需要经过多次变换才能得到的 RDD，此时虽然可以通过再次执行计算恢复该 RDD，但仍然要付出很大的代价。因此，Spark 提供了一类缓存算子，以帮助用户解决此类问题。

- **cache——缓存 RDD**

√ 算子函数格式：

```
-cache(): JavaRDD[T]
```

cache 将 RDD 的数据持久化存储在内存中，其实现方法是使用后面我们会介绍的

persist 算子。当需要反复使用某 RDD 时,使用 cache 缓存后,可以直接从内存中读出,不再需要执行该 RDD 的变换过程。需要注意的是,这种缓存方式虽然可以提高再次使用某个 RDD 的效率,但由于 cache 后的数据仅仅存储在内存中,因此不能解决 RDD 出错时需要再次恢复运算的问题。而且 cache 保存的数据在 Driver 关闭后会被清除,因此不能被在其他 Driver 中启动的 Spark 程序使用。

√ 示例:

cache 示例代码

```
1:  scala> val num = sc.parallelize(0 to 9,1)//构建 RDD
     num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at
parallelize at <console>:21
2:  scala> val result = num.map(x => x * x) //对原始 RDD 进行 map 变换
     result: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[8] at map
at <console>:23
3:  scala> result.cache //对新 RDD 进行缓存
    res19: result.type = MapPartitionsRDD[8]at map at <console>:23
4:  scala> result.count //统计新 RDD 中的元素个数
    res30: Long = 10
5:  scala> result.collect().mkString(",") //再次使用新 RDD,生成用逗号分隔的
序列
    res31: String = 0,1,4,9,16,25,36,49,64,81
```

示例代码建立了一个由数字 0 ~ 9 构成的原始 RDD,然后对原始 RDD 进行 map 变换,求得每个数字的平方,然后通过 cache 对新生成的 RDD 进行持久化。第 4 行统计新 RDD 的元素个数,第 5 行再次使用新 RDD 生成用逗号分隔的序列,此时 Spark 将直接访问持久化在内存中的新 RDD,而不需要再次进行之前的 map 变换。

- **checkpoint——建立 RDD 的检查点**

√ 算子函数格式:

```
- checkpoint(): Unit
```

对于需要很长时间才能计算出或者需要依赖很多其他 RDD 变化才能得到的 RDD,如果在计算过程中出错,要从头恢复需要付出很大的代价。此时,可以利用 checkpoint 建立中间过程的检查点,Spark 会将执行 checkpoint 操作的 RDD 持久化,以二进制文件的形式存放在指定的目录下。与 cache 不同的是,checkpoint 保存的数据在 Driver 关闭后仍然以文件的形式存在,因此可以被其他 Driver 中的 Spark 程序使用。

√ 示例：

checkpoint 示例代码

```
1:  scala>val rdd=sc.makeRDD(1 to 9,2) //构建原始 RDD
     rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at <console>:21
2:  scala> val flatMapRDD = rdd.flatMap(x => Seq(x,x)) //对原始 RDD 做 flatMap 变换
     flatMapRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at flatMap at <console>:23
3:  scala>sc.setCheckpointDir("my_checkpoint") //指定 checkpoint 存放的目录
4:  scala>flatMapRDD.checkpoint()//建立 checkpoint
5:  scala>flatMapRDD.dependencies.head.rdd //显示变换后 RDD 的依赖
     res2: org.apache.spark.rdd.RDD[_] = ParallelCollectionRDD[0] at makeRDD at <console>:21
6:  scala>flatMapRDD.collect() //显示变换后的 RDD
     res3:Array[Int]=Array(1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9)
7:  scala>flatMapRDD.dependencies.head.rdd //再次显示变换后 RDD 的依赖
     res4: org.apache.spark.rdd.RDD[_] = CheckpointRDD[2] at collect at <console>:26
```

示例代码的第 1、2 行，是对一个 1-9 的序列构成的 RDD 进行 flatMap 变换。在第 3 行指定了检查点所存放的目录。第 4 行建立检查点，可以看到此时并没有真正执行 checkpoint 操作。而且从第 5 行的输出可以看到，flatMap 变换后产生的 RDD 还是依赖于原始 RDD。在第 6 行使用 collect 变换显示新 RDD 的内容时，触发了 flatMap 变换的真正执行，此时输出结果中可以看到 checkpoint 也随之执行了。而且，从第 6 行的输出可以看到，新 RDD 的依赖已经随着 checkpoint 的执行而指向检查点数据了。在今后继续使用 flatMapRDD 时，如果 flatMapRDD 出现问题，对它的再次恢复就不需要由原始 RDD 进行 flatMap 变化的操作了，Spark 会直接从检查点数据中恢复 flatMapRDD。

- **persist——持久化 RDD**

√ 算子函数格式：

```
-persist(newLevel:StorageLevel):JavaRDD[T]
```

调用 persist 可对 RDD 进行持久化操作，利用参数 newLevel 可以指定不同的持久化方式，常用的持久化方式包括：

- MEMORY_ONLY：仅在内存中持久化，且将 RDD 作为非序列化的 Java 对象存储在 JVM 中。这种方式比较轻量，是默认的持久化方式。

- MEMORY_ONLY_SER：仅在内存中持久化，且将 RDD 作为序列化的 Java 对象存储(每个分区一个 byte 数组)。这种方式比 MEMORY_ONLY 方式要更加节省空间，但会耗费更多的 CPU 资源进行序列化操作。
- MEMORY_ONLY_2：仅在内存中持久化，且将数据复制到集群的两个节点中。
- MEMORY_AND_DISK：同时在内存和磁盘中持久化，且将 RDD 作为非序列化的 Java 对象存储。
- MEMORY_AND_DISK_SER：同时在内存和磁盘中持久化，且将 RDD 作为序列化的 Java 对象存储。
- MEMORY_AND_DISK_2：同时在内存和磁盘中持久化，且将数据复制到集群的两个节点中。

√ 示例：

persist 示例代码

```
1:  scala> val num = sc.parallelize(0 to 9,1) //构建 RDD
     num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
parallelize at <console>:12
2:  scala> num.getStorageLevel //显示 RDD 当前的持久化状态
     res8: org.apache.spark.storage.StorageLevel = StorageLevel(false,
false, false, false, 1)
3:  scala> num.persist() //使用 persist 进行默认的 MEMORY_ONLY 持久化
     res9: num.type = ParallelCollectionRDD[5]at parallelize at <console
>:21
4:  scala> num.getStorageLevel //显示 RDD 新的持久化状态
     res10: org.apache.spark.storage.StorageLevel = StorageLevel(false,
true, false, true, 1)
```

示例代码建立了一个数字 0 ~ 9 构成的 RDD，然后通过 persist 进行默认的 MEMORY_ONLY 持久化。

第5章

Spark 算法设计

在第 4 章,我们了解了 Spark 中的常用 RDD 算子,并结合一些简单的示例代码掌握了这些算子的功能和使用方法。我们可以看到,相比 MapReduce 编程模式中两个简单的 map 和 reduce 函数,Spark 提供了非常丰富的算子函数。这些丰富的算子函数,一方面形成了 Spark 的一个重要特色,就是对数据异常灵活的处理能力。这种灵活的处理能力,可以说是 Spark 除基于分布式内存机制实现的高性能特点外,另一个最强大的能力。但是,这种灵活性也像一把双刃剑,在提升 Spark 数据处理能力的同时,也设置了较高的学习门槛。为了便于大家掌握 Spark 算子函数,并能做到综合运用,在本章我们通过一些由简单到复杂的算法实例,为大家展示如何利用 Spark 丰富的算子函数设计和实现一些常用算法。

5.1 过滤

在数据挖掘的全部过程中,有一个非常重要的前置任务,就是数据的预处理。一方面,现实世界中采集的数据大多存在一些不完整或不一致的脏数据; 另一方面,某项具体的数据挖掘工作也不一定要用到全部数据进行处理。因此,原始数据在进入正式的数据挖掘流程之前,通常先通过数据清理、数据集成、数据变换、数据归约等操作进行预处理,以提高数据挖掘的性能和质量。在数据预处理的具体操作中,最常见和常用的操作即是过滤操作。过滤操作解决的目标问题是: 在原始数据中包含大量的记录,每条记录由某个实体及实体的若干属性构成,过滤操作的目标将符合一定条件的记录取出,在这过程中还可能进行格式转换。

在这里我们用一个简单的实例来说明使用 Spark 进行数据过滤的方法。我们的原始数据是一些网络流量日志文件,其内容如下。

accessLog.csv

```
0 18812345678 www.baidu.com/s?wd=spark
1 18823456789 www.cnblogs.com/jerrylead/
1 18834567890 www.qq.com/808/
0 13312345678 sina.com.cn/nba/
1 13323456789 weibo.com/csdnctoerwa
1 13334567890 163.com/15/0309
```

日志文件中的每行数据包含3列：第1列是接入网络的类型，0为2G或3G网络，1为4G网络；第2列是用户标识，例如手机号（当然，我们这里使用的是虚构的手机号）；第3列是用户访问的网站URL。因为我们最终的分析目标是要统计最新部署的4G网络中用户访问的网站数量情况，因此我们希望将使用4G网络访问Web的记录过滤出来，并将访问的网站URL仅保留网站的Host字段，以节省计算资源。因此，我们希望对accessLog.csv文件进行过滤处理后得到以下结果。

4GAccessLog.csv

```
18823456789 www.cnblogs.com
18834567890 qq.com
13323456789 weibo.com
13334567890 163.com
```

我们使用下面的代码完成以上过滤操作。

过滤算法代码

```
1: scala> val textRDD = sc.textFile("accessLog.csv", 2)
   textRDD: org.apache.spark.rdd.RDD[String] = inputFile MappedRDD[18]at
textFile at <console>:12
2: scala> val filteredRDD = textRDD.filter(_.split(" ")(0).equals("1"))
   filteredRDD: org.apache.spark.rdd.RDD[String] = FilteredRDD[19]at
filter at <console>:14
3: scala> val resultRDD = filteredRDD.map(word => {
        | val columns = word.split(" ")
        | val phoneNum = columns(1)
        | val host = columns(2).replaceAll("/.*", "")
        |phoneNum + " " + host
        | })
   resultRDD: org.apache.spark.rdd.RDD[String] = MappedRDD[20]at map at <
console>:16
4: scala> resultRDD.saveAsTextFile("4GAccessLog")
```

代码的执行过程如图 5-1 所示。代码第 1 行使用 sparkContext 类的 textFile 函数读取 accessLog. csv 文件生成两个分区的 Value 型 RDD 对象 textRDD。代码第 2 行使用 filter 算子对 textRDD 中的每条记录进行过滤。其中的 split 函数将每条记录按空格分隔进行切分，filter 算子会保留切分后第一个字段等于 1 的全部记录形成新的 RDD 对象 filteredRDD，其他记录则被丢弃。代码第 3 行对 filteredRDD 进行 map 变换。在这个 map 变换中进行了一系列操作，包括按空格切分每条记录，取出第 2 个字段（即用户手机号）、第 3 个字段（即 Host），并将 URL 中在“/ ”后的字符清空，得到 URL 中的 host。最后保存每个记录的用户手机号和访问的 host 保存为结果 resultRDD。代码第 4 行使用行动算子 saveAsTextFile 将结果存入文件夹 4GaccessLog 下。

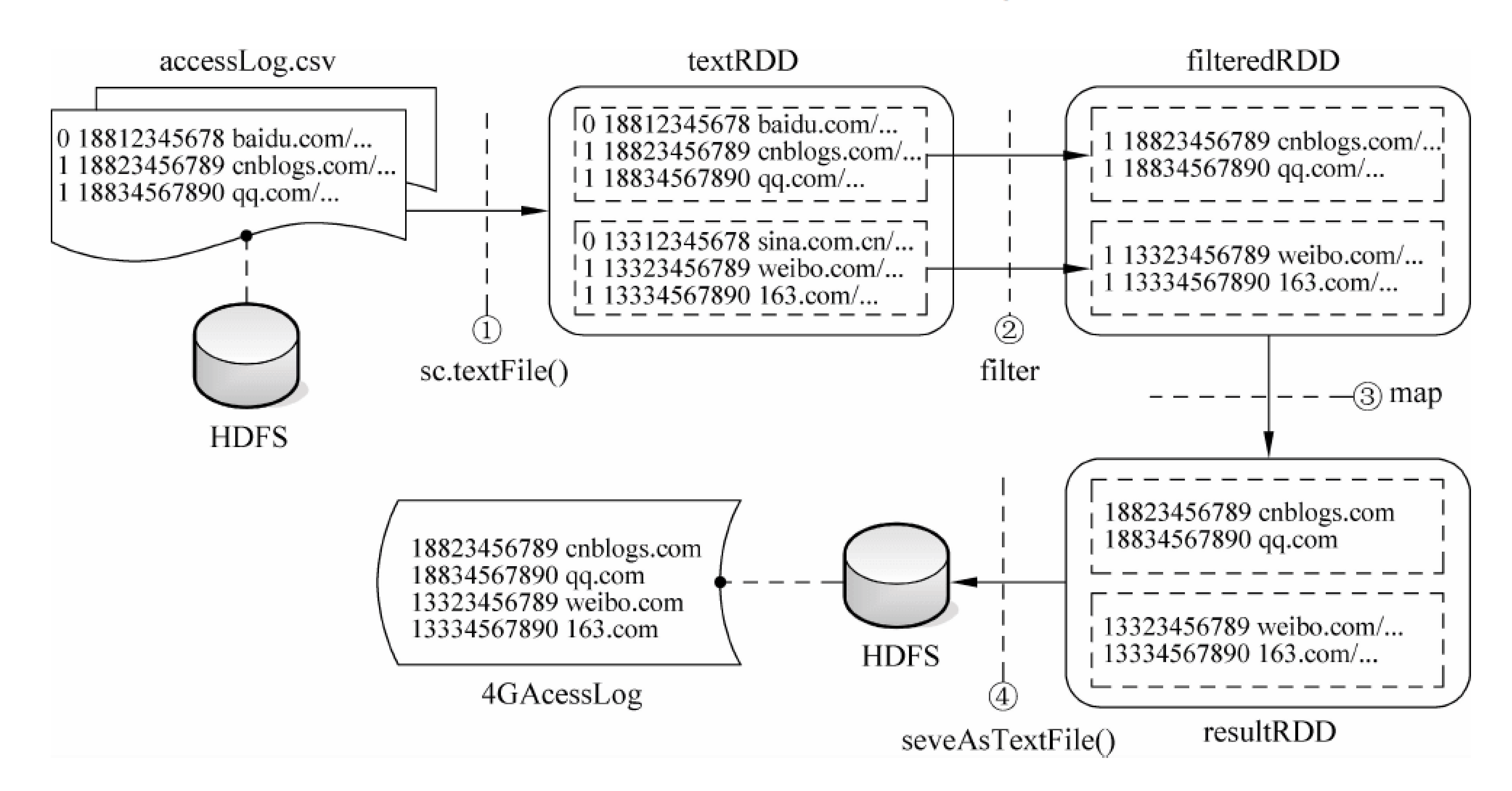


图 5-1 过滤算法过程

5.2 去重计数

在对数据进行了包括过滤在内的预处理后我们可以开始进行数据挖掘以获取有价值的信息。在数据挖掘工作中，计数是一类最基础的分析。我们熟知的 WordCount 程序完成的就是基础计数工作，即统计数据中某类实体的出现次数，例如文本文件中的单词、字母等。在这一节中我们讨论的是一类特殊的计数操作，去重计数[12]。去重计数解决的目标问题是：在原始数据中包含大量的记录，每条记录由某个实体及实体的若干属性构成，去重计数的目标是统计某一属性或属性组合 A 在另一关联属性或属性组合 B 不重复的情况下出现的次数。

我们以一个具体的实例来说明去重计数的功能。我们的输入数据是在上一节经

过过滤处理后的网络访问日志,即只保留了用户手机号和网站 Host 的日志文件,内容如下。

4GAccessLog.csv

```
18812345678 www.baidu.com
18812345678 www.baidu.com
18834567890 www.baidu.com
13312345678 www.baidu.com
13323456789 www.sina.com
13323456789 www.sina.com
```

文件中的每行数据包含 2 列: 第 1 列是户手机号,是用户的唯一标识; 第 2 列是用户访问的网站 Host。我们的目标是要统计每个网站 Host 被多少个不同的用户访问过,即希望得到如下的结果。

distinctUserCount.csv

```
www.baidu.com 3
www.sina.com 1
```

为此,我们编写了以下的代码以实现去重计数。

去重计数算法代码

```
1: scala> val textRDD = sc.textFile("4GAccessLog.csv")
   textRDD: org.apache.spark.rdd.RDD[String] = inputFile MapPartitionsRDD
[1]at textFile at <console>:23
2: scala> val mappedRDD = textRDD.map(word => {
    |val columns = word.split(" ")
    |val property1 = columns(0)
    |val property2 = columns(1)
    | (property2, property1)
    |})
   mappedRDD: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD
[2]at map at <console>:25
3: scala> val distinctedRDD = mappedRDD.distinct()
   distinctedRDD: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD
[5]at distinct at <console>:27
4: scala> val setOneRDD = distinctedRDD.mapValues(property1 => 1)
   setOneRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD
[6]at mapValues at <console>:29
5: scala> val reducedRDD = setOneRDD.reduceByKey(_+_)
   reducedRDD: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[7]
```

```
at reduceByKey at <console>:31
6:  scala> reducedRDD.map(x => x._1 + " " + x._2).saveAsTextFile("distinct
UserCount")
```

代码的执行过程如图 5-2 所示。代码第 1 行使用 sparkContext 类的 textFile 函数读取 4GAaccessLog.csv 文件生成 Value 型 RDD 对象 textRDD。代码第 2 行将 textRDD 中元素按空格分割后，把第 1 列(property1，即用户手机号)设为 Value，把第 3 列(property2，即网站 Host)设为 Key，构成 Key/ Value 型的 RDD 对象 mappedRDD。代码第 3 行使用 distinct 算子对 mappedRDD 中的键值对进行去重，构成新的 Key/ Value 型的 RDD 对象 distinctedRDD。代码第 4 行使用 mapValues 算子把去重后得到的 distinctedRDD 中每条记录的 Value 设为 1，Key 保持不变，得到新的 Key/ Value 型的 RDD 对象 setOneRDD。代码第 5 行使用 reduceByKey 算子统计 setOneRDD 中每个 Key 值(即网站 Host)的数量，即为我们希望统计的结果。最后，代码第 6 行使用行动算子 saveAsTextFile 将结果存入文件夹 distinctUserCount。

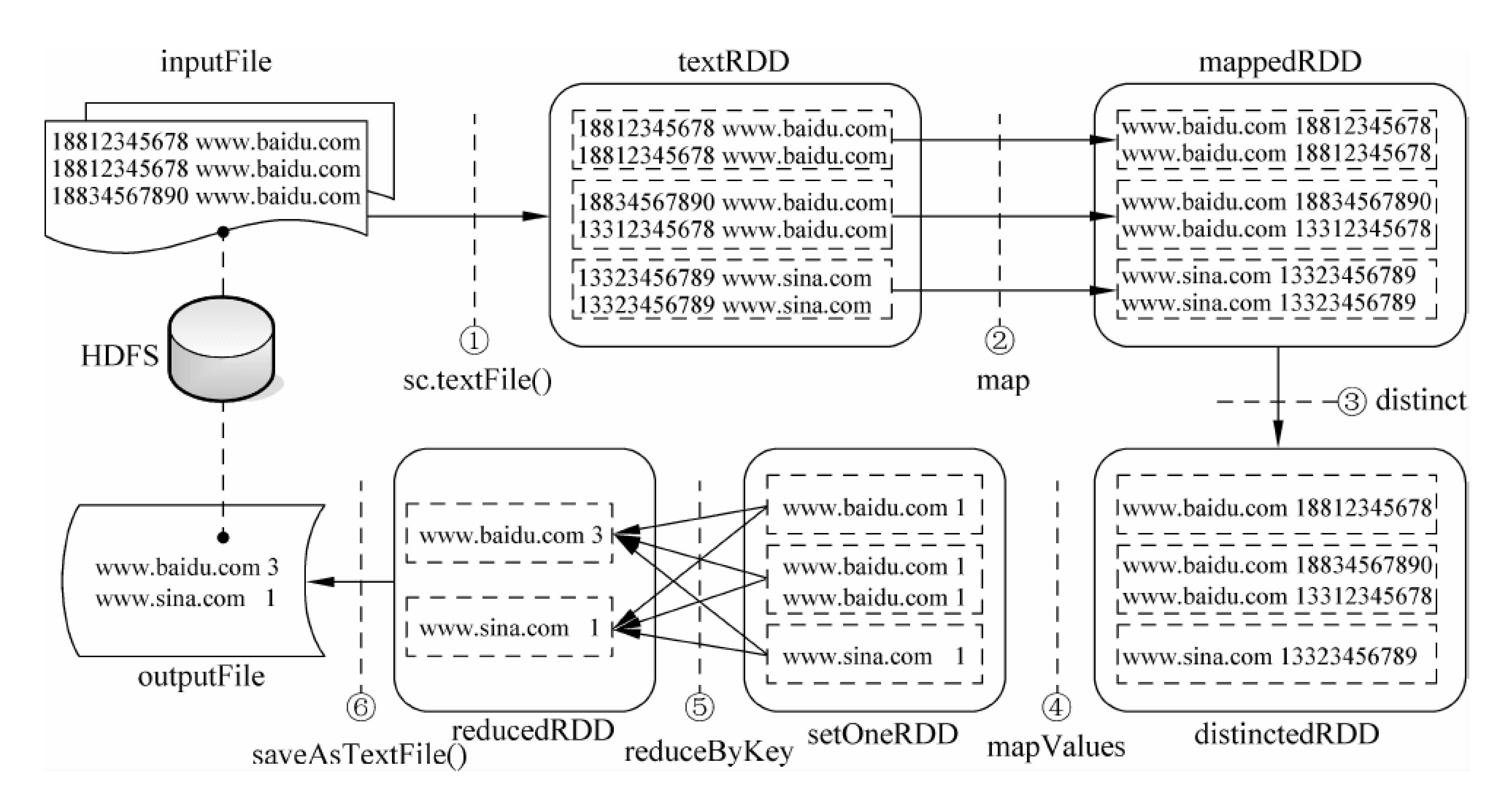


图 5-2 去重计数算法过程

5.3 相关计数

相关计数是数据挖掘工作中的一类经典基础问题，即计算两个或多个实体以一定方式共同出现的次数或概率。例如在基于统计学的自然语言处理领域，有一类经典模型是概率语言模型(Probabilistic Language Models)[13][14]。与基于明确的语言规则对文

本进行分析的传统方法不同，概率语言模型将文本看成是一系列服从一定概率分布的词项的样本集合，从而引入以概率估计为核心的数学方法对文本进行分词、主题提取等各种分析。在概率语言模型中，最核心基础的算法就是计算文本中不同词之间的共同出现次数。依问题和场景的不同，这些统计方式可能是两个词，也可能是多个词，可能是有序的，也可能是无序的。为简单起见，我们这里以计算文本中两个词共同出现的次数为例来介绍相关计数的 Spark 算法。假设我们有一个简单的文本文件如下所示。

textFile.csv

```
It was the best of times
It was the worst of times
```

我们希望统计这个文本文件中两个单词共同出现的次数，即希望得到如下的结果。

outputFile.csv

```
(the,worst) 1
(of,times) 2
(best,of) 1
(the,best) 1
(was,the) 2
(It,was) 2
(worst,of) 1
```

为达成这一分析目标，我们设计实现了如下代码。

相关计数算法代码

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object CrossCorrelation {
def main(args: Array[String]) {
  if (args.length != 2) {
    System.err.println("Usage: CrossCorrelationDemo <input path> <output path>")
    System.exit(1)
  }
```

```
10:     val inputFile = args(0)
11:     val outputFile = args(1)
12:     val conf = new SparkConf().setAppName("CrossCorrelationDemo")
13:     val sc = new SparkContext(conf)
14:     val textRDD = sc.textFile(args(0)).cache
15:     val wordRDD = textRDD.map(_.split(" "))
16:     val mappedRDD = wordRDD.flatMap(words =>{
17:       for(i <-0 until words.length -1)
18:       yield ((words(i), words(i +1)),1)
19:     })
20:     val reducedRDD = mappedRDD.reduceByKey(_+_)
21:       reducedRDD.map(x =>x._1 + " " +x._2).saveAsTextFile(outputFile)
22:   }
23: }
```

因为我们编写了一个独立运行的 Spark 程序，因此我们可以首先将代码打包成 jar 文件，然后我们可以在 Shell 中使用如下的命令执行这段代码对 textFile. csv 进行处理。

```
spark-submit \
  --class CrossCorrelation\
  --master local\
  /path/CrossCorrelation.jar\
  /path/input \
  /path/output
```

class 参数指明了应用程序的启动类，master 参数指明了集群 master 节点的地址，由于本例比较简单，因此我们指定 master 参数为 local，即使用 1 个 worker 线程在本地运行该 spark 应用程序。接下来需要指定应用程序的 jar 包地址以及输入输出文件的路径。

代码执行的过程如图 5-3 所示。因为这段代码是一个独立 Spark 程序，而不是直接在 Spark Shell 中运行的脚本。因此，需要在代码第 12 行设置 Spark 参数，并在第 13 行创建 SparkContext 实例。代码第 14 行读取参数指定的 textFile. csv 本内容，并创建 Value 型 RDD 对象 textRDD。代码第 15 行以空白字符分割输入文本，得到所有输入单词，并保存在 Value 型 RDD 对象 wordRDD 中。代码第 16 行对 wordRDD 进行 flatMap 变换，以生成以相邻单词对为 Key、次数为 Value 的 Key/ Value 型 RDD 对象 mappedRDD。代码第 20 行使用 reduceByKey 算子对 mappedRDD 进行简单的相同 Key 值次数相加，即得到了单词对出现的次数。最后在代码第 21 行保存结果。

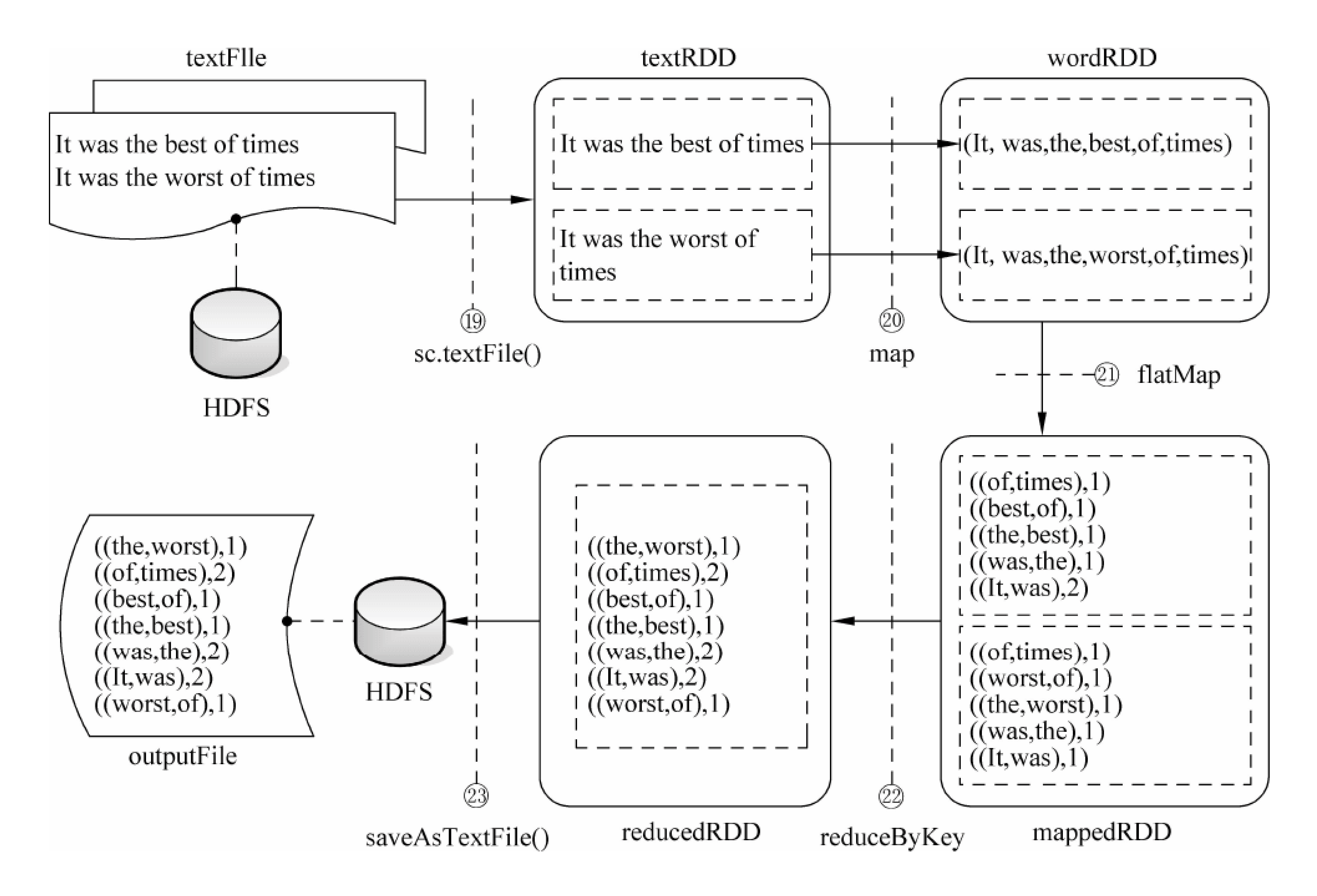


图5-3 相关计数算法过程

5.4 相关系数

现实世界是由很多事物和现象构成的，在这些事物和现象间，存在着错综复杂的相关关系，例如天气晴雨与天空云量、气压等变量间的关系，孩子身高与父母身高、饮食习惯等变量间的关系。为了研究这些变量间的关系，统计学家们提出了相关系数（correlation coefficient）这一数学指标，用以反映变量之间相关关系密切程度。常见的相关系数有皮尔逊（pearson）相关系数[15]、斯皮尔曼等级（spearman's rank）相关系数[16]、肯德尔等级（Kendall tau rank）相关系数[17]等。其中，皮尔逊相关系数可用于度量两个变量 X 和 Y 之间的线性相关关系，其值介于 -1 与 1 之间，由于其简单有效的特性而被广泛应用。两个变量之间的皮尔逊相关系数定义为两个变量之间的协方差和标准差的商，计算公式如下。

$$\rho_{X,Y}=\frac{\mathrm{cov}(X,Y)}{\delta_X\delta_Y} \tag{5-1}$$

下面我们以一个实例来说明采用Spark计算皮尔逊相关系数的方法。假设我们有一份如下的病人基因数据。

geneFile.csv

```
g1 r2 p1 1.86
g2 r2 p1 0.74
g3 r2 p1 1.24
g1 r2 p2 2.46
g2 r2 p2 3.24
g3 r1 p2 1.44
g3 r2 p2 1.23
g1 r2 p3 1.55
g2 r1 p3 0.76
g2 r2 p3 1.34
g3 r2 p3 1.45
g1 r2 p4 1.56
g2 r2 p4 1.33
g3 r2 p4 1.45
g2 r2 p1 1.34
g1 r2 p3 1.23
g2 r4 p4 1.33
g1 r2 p2 1.55
g2 r2 p3 1.65
```

文件中包含病人的基因数据，每行是一个记录，记录各字段用空格分隔，4 个字段含义为：基因 ID、reference 值、病人 ID、生物标记(Double 形式)。我们的目标是计算 reference 值为 r2 的各基因之间的皮尔逊相关系数，每个基因 ID 变量都以 4 个病人(p1-p4)为实验对象，为了实验的准确性，有可能对同一个人做不止一次的生物实验检测，我们使用多次试验的平均生物标记值作为该用户的测量值，最终希望得到如下结果。

outputFile.csv

```
(g2,g3) -0.45212493718074226
(g1,g3) -0.9516745980717424
(g1,g2) 0.5741067710236532
```

为了实现以上计算目标，我们编写了如下代码。

皮尔逊相关系数计算代码

```
import org.apache.commons.math3.stat.correlation.PearsonsCorrelation
import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.SparkContext._
import scala.collection.mutable
```

```
import scala.collection.mutable.ArrayBuffer
object PearsonsCorrelationDemo {
  def main(args: Array[String]) {
    if (args.length < 3) {
       System.err.println("Usage: PearsonsCorrelation <reference> <input> <output>")
      System.exit(1)
      }
    val reference = args(0)
    val inputdir = args(1)
    val outputdir = args(2)
    val conf = new SparkConf().setAppName("pearson corrlation")
    val sc = new SparkContext(conf)
    val REF = sc.broadcast(reference)
    val text = sc.textFile(inputdir)
    val filteredRDD = text.filter(line => line.contains(REF.value))
    val pairRDD = filteredRDD.map(line => line.split(" ")).map(a => (a(0), (a(2), a(3))))
    val groupRDD = pairRDD.groupByKey()
    val cartRDD = groupRDD.cartesian(groupRDD)
    val sortedRDD = cartRDD.filter(pair => pair._1._1 < pair._2._1)
    val mappedRDD = sortedRDD.map(x => ((x._1._1, convertToMap(x._1._2)),
      (x._2._1, convertToMap(x._2._2))))
      val corRDD = mappedRDD.map(x => ((x._1._1, x._2._1), caculateCorrelation(x._1._2, x._2._2)))
    corRDD.map(x => x._1 + " " + x._2).saveAsTextFile(outputdir)
  }
    def convertToMap(tuples: collection.Iterable[Tuple2[String, String]]) = {
    var map = new mutable.HashMap[String, MutableDouble]()
      for (tuple2: Tuple2[String, String] <- tuples) {
      if (map.contains(tuple2._1)) {
        map(tuple2._1).increase(tuple2._2.toDouble)
      } else {
        map.put(tuple2._1, new MutableDouble(tuple2._2.toDouble))
      }
    }
    map
  }
      def caculateCorrelation(hashmap1: mutable.HashMap[String, MutableDouble], hashmap2: mutable.HashMap[String, MutableDouble]): Double = {
    var x = new ArrayBuffer[Double]()
    var y = new ArrayBuffer[Double]()
    for ((key, value) <- hashmap1) {
      if (hashmap2.contains(key)) {
        x += value.average()
        y += hashmap2(key).average()
      }
```

```
48:       }
49:       if (x.size < 3) {
50:         Double.NaN
51:       } else {
52:         val ps = new PearsonsCorrelation()
53:         var corr = ps.correlation(x.toArray, y.toArray)
54:         corr
55:       }
56:     }
57:   }
58:   class MutableDouble extends Serializable{
59:     def this(d:Double){
60:       this()
61:       value += d
62:       count = 1
63:     }
64:     private var value = 0.0
65:     private var count = 0
66:     def increase(d:Double): Unit = {
67:       value += d
68:       count += 1
69:     }
70:     def average():Double = {
71:         value/count
72:     }
73:   }
```

代码的运行过程如图 5-4 所示。代码第 18 行首先使用 sparkContext 类的 textFile 函数读取 geneFile.csv 文件生成一个 Value 型 RDD 对象 textRDD。代码第 19 行调用 filter 函数过滤出 reference 为 r2 的记录,得到 filterRDD,为了增加程序的可扩展性,使得程序可以根据不同的 reference 值进行相关系数计算,我们将 reference 值作为程序一个输入参数,并将该值广播到各个计算节点,广播操作由代码的第 17 行完成,各个计算节点根据输入的参数可以过滤出不同的数据。然后代码第 20 行调用 map 函数对 filterRDD 中的每一条记录进行格式转换,转换为 key-vlaue 形式,其中 Key 为基因 ID,value 为(病人 ID、生物标记),得到 pairRDD。代码第 21 行调用 groupByKey 函数,对 pairRDD 按 key 即基因 ID 进行汇聚,得到 groupRDD。由于要计算所有基因 ID 两两之间的相关系数,因此代码第 22 行对汇聚完的数据调用 cartesian 函数求笛卡尔乘积得到 cartRDD,例如 Gene-ID 集合为{g1,g2,g3},那么需要求出(g1,g2),(g1,g3),(g2,g3)之间的相关系数。我们求笛卡尔乘积可以得到:

(g1,g1),(g1,g2),(g1,g3),(g2,g1),(g2,g2),(g2,g3),(g3,g1),(g3,g2), (g3,g3)

由于(g_i,g_j),(g_j,g_i)的相关系数相等,且(g_i,g_i)没有意义,因此代码第 23 行对笛

卡尔乘积的结果调用 filter 函数进行了过滤，以保留集合(g_i, g_j)(其中 $i<j$)，最终保留得到 sortedRDD。集合(g_i, g_j)中不同的变量对应的 value 值为(病人 ID、生物标记)集合。代码第 24 行使用 map 算子对上一步结果中的每一行进行操作。在 map 算子内调用代码 29 ~ 39 行定义的 convertToMap 函数将变量 g_i、g_j对应的 value 值放入不同的 hashmap 中。hashmap 的 key 为病人 ID，value 值为代码 58 ~ 72 行定义的 MutableDouble 类型值。MutableDouble 类包括两个方法：①increase( )方法，因为需要计算同一个病人 ID 的生物标记的平均值，因此 increase( )方法负责记录生物标记值的总和(存储为 value)和样本数(存储为 count)；②average( )方法计算同一个病人 ID 的生物标记的平均值。convertToMap 函数首先将 g_i 对应的所有 value 值放入 hashmap1 中，convertToMap 函数对 g_j对应的 value 值也进行相同的操作，将 g_j对应的 value 值放入 hashmap2。利用 hashmap1 和 hashmap2 构建了如表 5-1 所示的数据集。

表 5-1 hashmap 构建的数据集

Key	**hashmap1**	**hashmap2**
	Value	**Value**
Patient-ID1	(sum(values), count)	(sum(values), count)
Patient-ID2	(sum(values), count)	(sum(values), count)
Patient-ID3	(sum(values), count)	(sum(values), count)
…	…	…

然后代码的 24 行使用 map 算子对数据进行操作，map 算子调用 40 ~ 57 行定义的 caculateCorrelation 的函数计算变量之间的相关系数，在 caculateCorrelation 函数中 35、36 行首先创建两个数组来存储两个变量 g_i 和 g_j的值，43 ~ 48 行将同一用户的生物标记的平均价值放入两个数组的对应位置，构建了如表 5-2 所示的数据集。

表 5-2 数组构建的数据集

	Patient-ID1	**Patient-ID2**	**Patient-ID3**	…
g_i	avg(values)	avg(values)	avg(values)	…
g_j	avg(values)	avg(values)	avg(values)	…

由于皮尔逊相关系数需要样本的数量不少于 3 个，因此 49、50 行判断样本数量是否不少于 3 个，如果样本数量不少于 3 个则代码 51 ~ 55 行调用 org. apache. commons 的 PearsonsCorrelation 计算两个变量的相关系数。

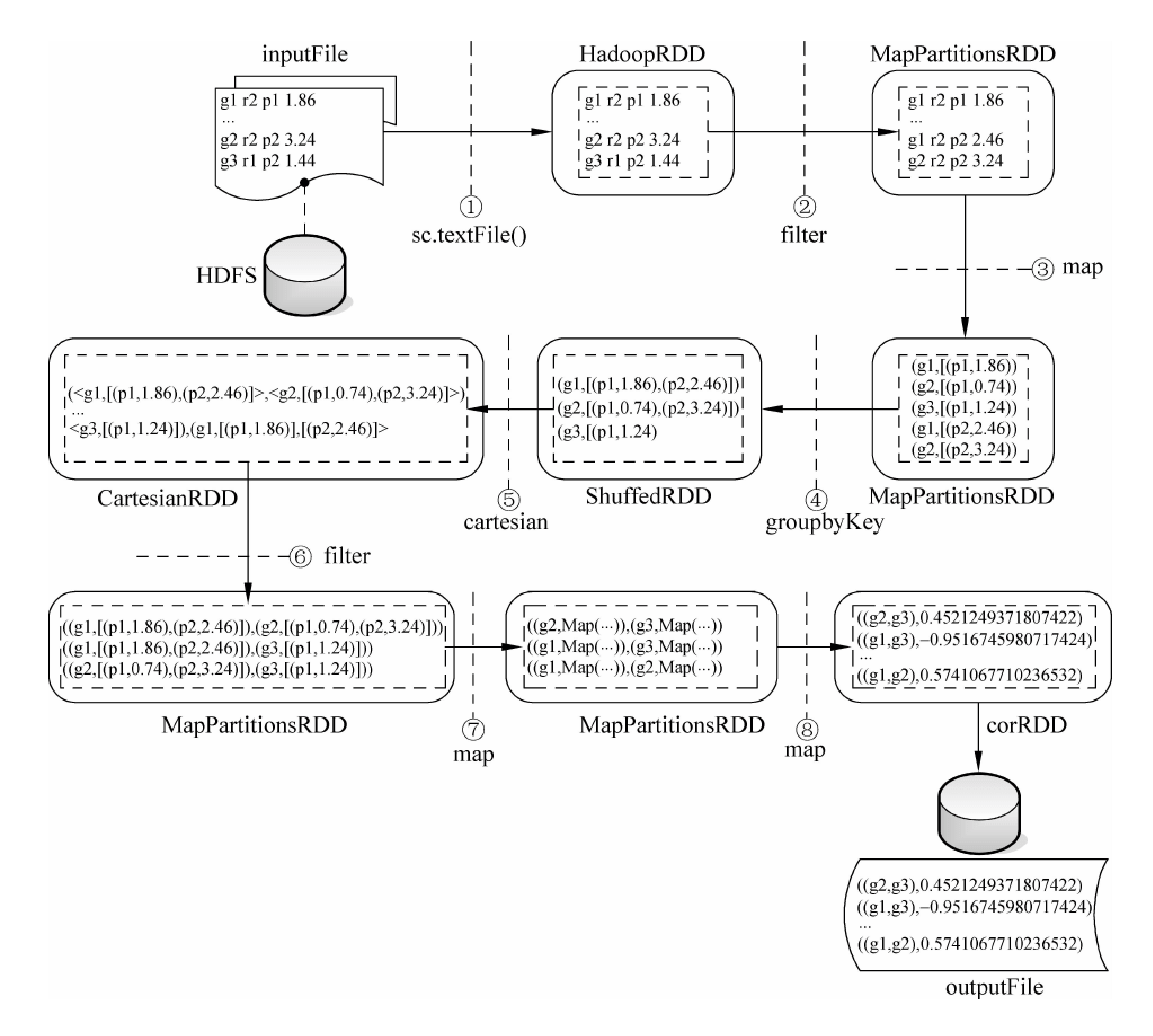


图 5-4 相关系数算法过程

5.5 数据联结

在传统的数据库系统中，数据联结是一个基本且常用的操作，例如在数据库中存在两张不同的表，一张表记录了用户的访问信息包括用户 ID、用户访问时间、用户访问次数等，另一张表则记录了用户的购买信息包括用户 ID、用户购买商品、商品价格、购买次数等。现在需要将两张表进行联结操作，以便全面了解一个用户的全部信息，这时就需要对两张表进行联结操作。两张表的联结需要指定一个联结条件列，即如果该列的值在两表中相同则进行联结操作，将两张表的内容合并。如果联结条件列的值在两张表中不同，数据库可以提供多种方式来决定是否将记录保留。大多数的数据库都会提供 join、leftOuterJoin、righOuterJoin 表联结操作。表联结 join 操作只有联结条件值在两个表中都出现时才合并输出。表联结 leftOuterJoin、righOuterJoin 操作除了将联结

条件值相同的项合并输出外，还会将其中一个表的全部记录保留，需要保留的表取决于输入两个表的位置，将操作中首先输入的表称为左表，后输入的表称为右表，leftOuterJoin 将左表的全部数据保留，而 righOuterJoin 则将右表的全部数据保留。

在 Spark 中也提供了类似于数据库表联结的数据联结操作[18]。Spark 的数据联结操作针对的是 Key/Value 型 RDD 数据。Key/Value 型对 RDD 中的键相当于数据库中的联结条件列。Spark 联结两个 Key/Value 型 RDD 时，根据两个 RDD 的键进行联结条件判断。下面我们以一个实例来说明 Spark 中的各种数据联结操作。假设我们有两个如下的数据列表，表中有两列，一列为字母列，一列为数据值列。

```
inputList1.csv
```

```
a 0
a 1
b 2
c 5
d 6
```

```
inputList2.csv
```

```
a 1
b 3
c 6
e 2
```

我们需要得到不同联结结果。

- 期望 1：如果两个表中的字母列的值相同，则将两个表中对应行的数据列进行联结；如果字母只在一个表中出现，则该行记录被舍弃，即希望得到如下结果。

```
outputFile1.csv
```

```
a (0,1)
a (1,1)
b (2,3)
c (5,6)
```

- 期望 2：如果两个表中的字母列的值相同，则将两个表中对应行的数据列进行联结，并且保留 inputList1 中的所有数据，即希望得到如下结果。

outputFile2.csv

```
d (6,None)
b (2,Some(3))
a (0,Some(1))
a (1,Some(1))
c (5,Some(6))
```

- 期望 3：如果两个表中的字母列的值相同，则将两个表中对应行的数据列进行联结，并且保留 inputList2 中的所有数据，即希望得到如下结果。

outputFile3.csv

```
b (Some(2),3)
e (None,2)
a (Some(0),1)
a (Some(1),1)
c (Some(5),6)
```

- 期望 4：如果想将两个表的全部数据保留，并根据字母列将两个表的数据联结，即希望得到如下结果。

outputFile4.csv

```
d {(6),(6)}
b {(2),(2)}
e {(),()}
a {(0,1),(0,1)}
c {(5),(5)}
```

为了实现以上 4 个联结操作，我们编写了如下代码。

联结算法代码

```
1:  scala> val rddA = sc.parallelize(List(('a',0),('a',1),('b',2),('c',5),('d',
6)))
    rdd_1: org.apache.spark.rdd.RDD[(Char, Int)] = ParallelCollectionRDD
[0]at parallelize at <console>:21
2:  scala> val rddB = sc.parallelize(List(('a',1),('b',3),('c',6),('e',2)))
    rdd_2: org.apache.spark.rdd.RDD[(Char, Int)] = ParallelCollectionRDD
[1]at parallelize at <console>:21
3:  scala > rddA.join(rddB).map(x => x._1 + " " + x._2).saveAsTextFile
("outputFile1")
```

```
4:  scala > rddA. leftOuterJoin (rddB). map (x => x._1 + " " + x._2).
saveAsTextFile("outputFile2")
5:  scala > rddA. rightOuterJoin (rddB). map (x => x._1 + " " + x._2).
saveAsTextFile("outputFile3")
6:  scala > rddA.cogroup (rddB).map (x => x._1 + "{(" + x._2._1.toArray.
mkString(",") + "),("+x._2._1.toArray.
    mkString(",") + ")}").saveAsTextFile("outputFile4")
```

代码执行的过程如图 5-5 所示。代码第 1、2 行使用 sparkContext 类的 parallelize 将 List 表集合进行并行化,生成 Key/ Value 型 RDD 对象 rddA 和 rddB。代码第 3 行调用 Key/ Value 型 RDD 的转化算子 join,内联结 rddA 和 rddB,并将结果收集输出得到期望 1 的结果。代码第 4 行调用 Key/ Value 型 RDD 的转化算子 leftOuterJoin,左外联结 rddA 和 rddB,并将结果收集输出得到期望 2 的结果。代码第 5 行调用 Key/ Value 型 RDD 的转化算子 rightOuterJoin,右外联结 rddA 和 rddB,并将结果收集输出得到期望 3 的结果。代码第 6 行调用 Key/ Value 型 RDD 的转化算子 cogroup,联结 rddA 和 rddB 求出两个 RDD 数据的并集,并将结果收集输出得到期望 4 的结果。

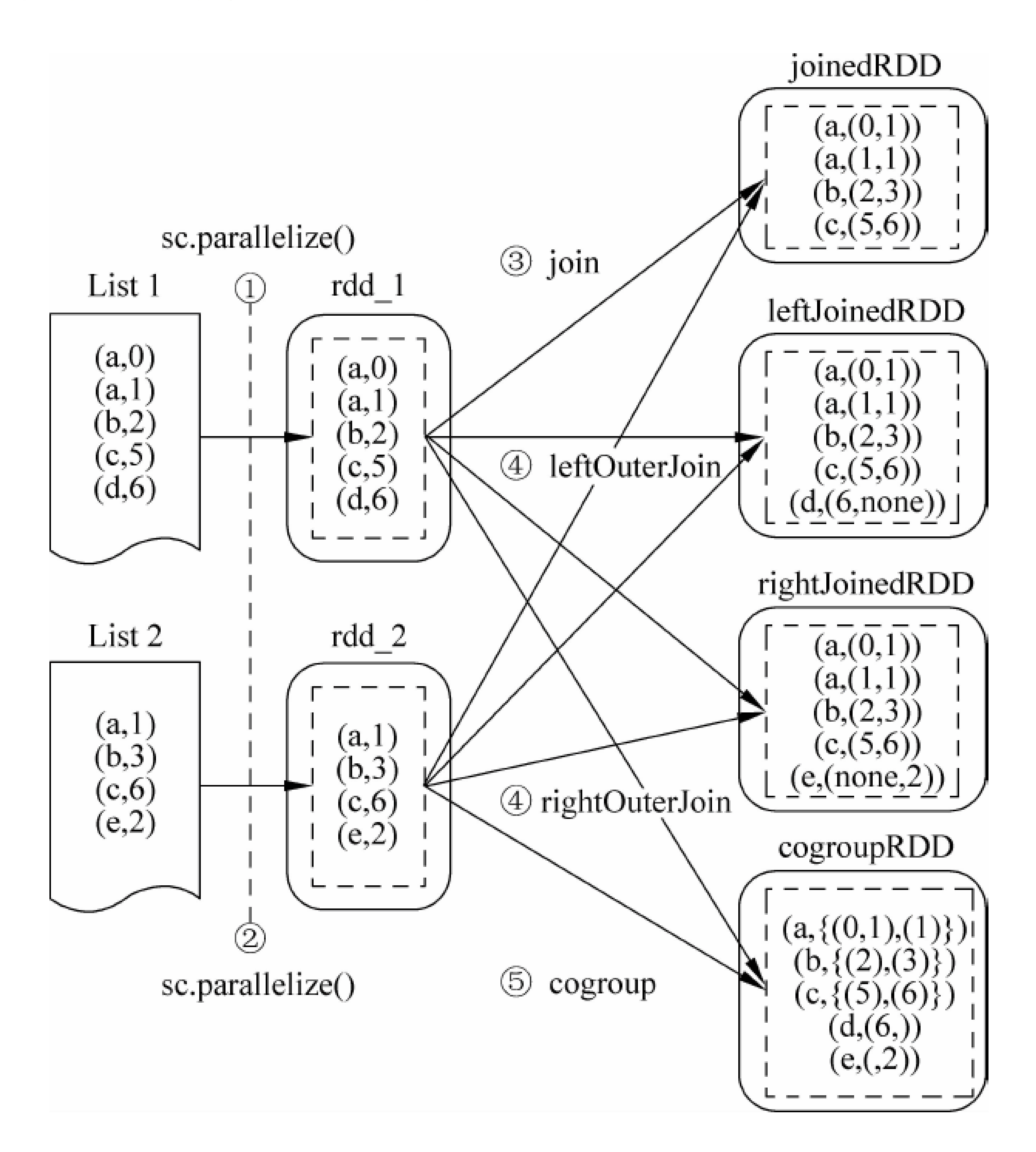


图 5-5 联结算法过程

5.6 Top-K

在现实统计任务中,统计观测数据中排名 Top-K[19]的变量是一个极其常见的统计任务。统计变量排名 Top-K 主要出于两方面的原因:一方面,现实世界中采集的数据大多数量很大,想对数据的全部值进行统计分析、建模等工作需要大量的计算工作,很难实现;另一方面,现实中许多变量都服从幂律分布,即大部分的资源都被排名靠前的少数变量所占有。例如,在进行网络流量统计时,超过 50% 的网络流量都被几个大网站(例如 QQ、百度、新浪等)所占用。为了简化分析任务的难度而又不失去分析的准确性,网络运营者更加关心排名靠前(例如前 K 个,简称为 Top-K)的网站,然后根据 Top-K 网站的流量情况来合理规划和调整网络。Top-K 算法解决的目标问题是:在原始数据中包含大量的记录,对记录的某一特征值进行统计排序,最终只取出排名前 K 个的记录。这种问题最简单的方法就是将整个数据逐个进行排序,找出排名前 K 个的记录,但这种方法显然效率不高。我们使用并行的思想处理该问题,采用分而治之的思想。我们将全部数据划分为多个分区,先在每个分区上计算分区上的 Top-K,然后将分区合并找到整个数据的 Top-k。下面我们以一个具体的实例来说明 Top-K 算法的 Spark 实现。我们的输入数据是一个包含许多文档的文件集数据,其部分内容如下所示。

```
inputFile
```

```
Apache Spark is an open source cluster computing framework originally
developed in the AMPLab at University of California, Berkeley but was
later donated to the Apache Software Foundation where it remains today. In
contrast to Hadoop's two - stage disk - based MapReduce paradigm, Spark's
multi - stage in - memory primitives provides performance up to 100 times
faster for certain applications. By allowing user programs
to load data into a cluster's memory and query it repeatedly, Spark is well
 - suited to machine learning ……
```

出于篇幅所限我们没有列出所有文件内容,只是显示了其中的一部分内容。文档中的每个单词间以空格分隔。我们的目标是要统计每个单词的出现次数,最终找出出现次数最高的前 5 个单词,即希望得到如下的结果。

outputFile.csv

```
Spark 45
and 34
a 28
the 25
Apache 22
```

为此,我们编写了如下代码以实现 Top-K 算法。

TopK 算法代码

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD
import scala.reflect.ClassTag
import scala.collection.generic.Growable
import java.util.PriorityQueue
import scala.collection.JavaConverters._
object TopK {
    def main(args: Array[String]): Unit = {
        val Array(input, output) = args//输入与输出路径
        val sc = new SparkContext(new SparkConf())
        val K = 5 //指定 K 的大小
        val text = sc.textFile(input)
        val count = text.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_ + _) //统计单词频数
        implicit val ord = Ordering.by[(String, Int), Int](_._2) //指定单词频数作为排序规则
        val topkResult = topK(count, K) //统计频数最高的前 K 个单词
         sc.parallelize(topkResult,1).map(x => x._1 + " " + x._2).saveAsTextFile(output);
        sc.stop()
    }
    def topK[T: ClassTag](rdd: RDD[T], K: Int)(implicit ord: Ordering[T]): Array[T] = {
        if (K==0) {
            Array.empty
        } else { //使用 BoundedPriorityQueue 统计每个分区的 TopK
            val topKRDDs = rdd.mapPartitions(items => {
                val queue = new BoundedPriorityQueue[T](K)
                items.foreach(queue +=_)
                Iterator.single(queue)
            })
            if (topKRDDs.isEmpty()) {
                Array.empty
            } else { //将每个分区的 TopK 进行归并
```

```
32:                 val reducedTopK = topKRDDs.reduce(_ ++= _)
33:                 reducedTopK.toArray.sorted(ord.reverse) //将最终 TopK 进行递减排序
34:             }
35:         }
36:     }
37: }
38: class BoundedPriorityQueue[T](maxSize: Int)(implicit ord: Ordering[T]) extends Iterable[T]with
39: Growable[T]with Serializable { //有界优先队列
40:     private val queue = new PriorityQueue[T](maxSize, ord) //使用 java.util.PriorityQueue
41:     override def iterator: Iterator[T] = queue.iterator.asScala //获取队列迭代器
42:     override def size: Int = queue.size
43:     override def += (elem: T): this.type = {
44:         if (size < maxSize) { //当前队列长度小于最大长度,直接向队列中添加元素
45:             queue.add(elem)
46:         } else { //当前队列长度已经达到最大长度
47:             val head = queue.peek()
48:             if (ord.gt(elem, head)) { //添加的元素大于最小元素,则进行替换
49:                 queue.poll
50:                 queue.add(elem)
51:             }
52:         }
53:     this
54:   }
55:   override def clear = queue.clear
56: }
```

代码的执行过程如图 5-6 所示。代码的第 10 行是程序的输入输出路径。代码的第 4 行创建一个 SparkContext 实例。代码第 12 行设置 Top-K 的 K 值,由于我们只需最终保留词频排名最多的前 5 名单词,因此这里设置 K 等于 5。代码第 6 行读取参数指定的输入文本内容,并创建 RDD 对象。代码的第 14 行是一个典型的 wordcount 代码,首先使用 split 方法将文档每一行按空格分隔成单词,然后使用 map 算子将每一个单词转化为 Key/ Value 型 RDD,Key 为单词,Value 值为 1。最后调用 reduceByKey 算子统计每个单词的频次。由于代码的第 14 行输出为一个 Key/ Value 的 RDD,因此我们需要指定数据按那个值进行排序操作。在这里我们是对频次进行排序,因此我们指定以 RDD 中的第 2 列值也就是 Value 值进行排序。代码的第 15 行使用 scala 语言中的 Ordering. by 方法来定义排序的模型,以输入变量的第二个值为排序依据。后续代码会调用该排序模型。代码第 16 行统计出整个数据中词频排名前 5 的单词,该行代码调

用了代码 20 ~ 37 行自定义的 topK 方法。topK 方法分为两部分，第一部分代码 21 ~ 28 行，实现统计每个分区上单词词频 Top-K 的单词。在每个分区的统计上，程序调用了 38 ~ 56 行定义的有界优先队列，有界优先队列是一种数据排序的数据结构，该结构首先判断当前队列长度是否小于指定的队列最大长度，如果队列未达到指定的队列最大长度则直接向队列中添加元素，如果当前队列长度已经达到指定的队列最大长度，则判断被添加的元素是否大于队列中最小的元素，如果大于最小元素则用该元素替换队列中的最小元素。我们使用该数据结构来快速的统计、存储每个分区上的词频 Top-K 单词。topK 方法的第二部分代码 31 ~ 34 行将每个分区的 TopK 单词进行归并求出整个数据集的词频 Top-K 单词，即为我们希望得到的结果。

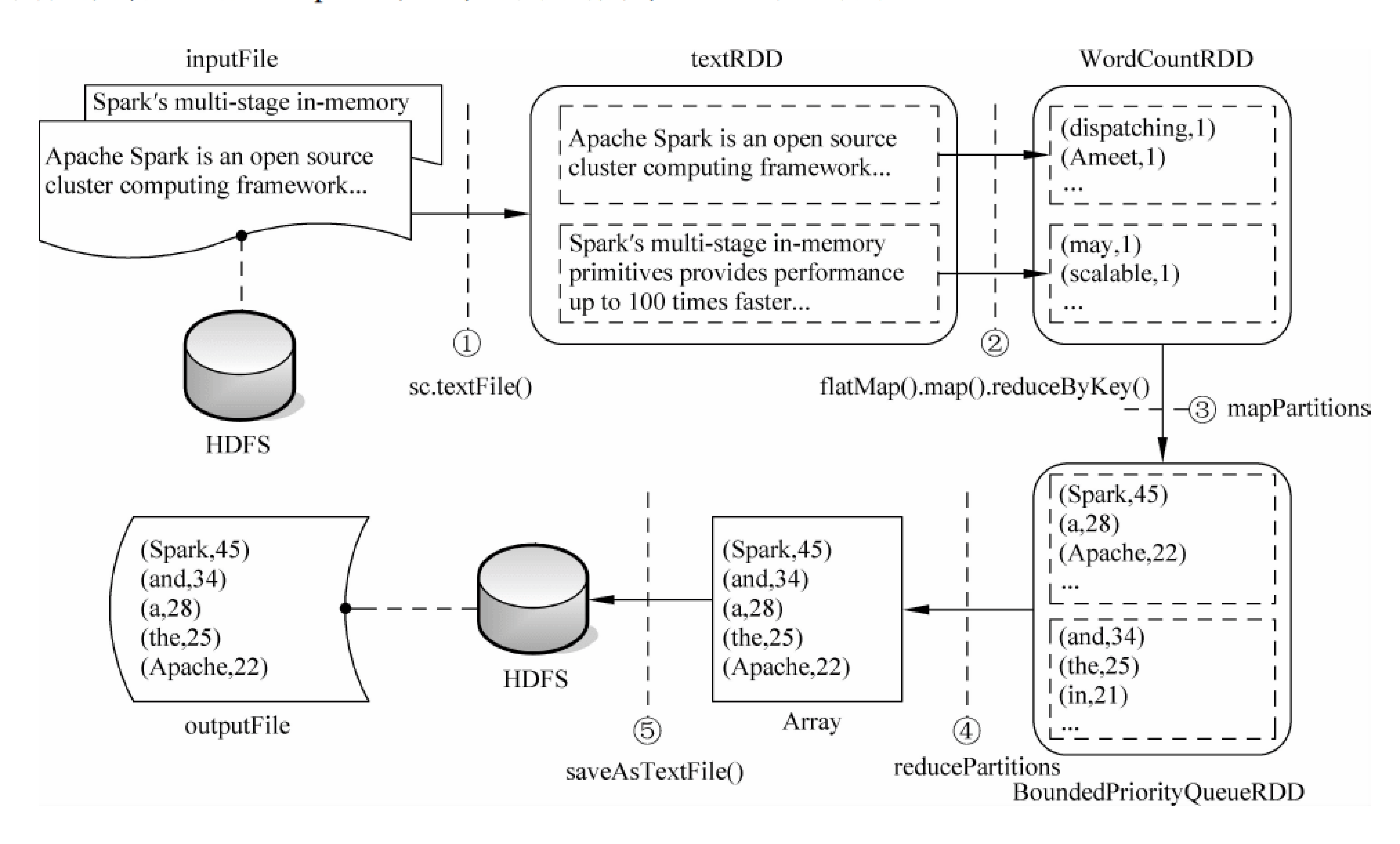


图 5-6 TopK 算法过程

5.7 K-means

K-means[20][21] 算法是数据挖掘算法中非常典型的一个聚类算法。K-means 算法中的 K 指的是希望聚类结果生成的类别个数。算法将 n 个数据对象聚合为 K 个类别以使得在同一类中的对象相似度较高，而不同类中的对象相似度较小。相似度通常以对象到类质心的距离作为相似性的评价指标。K-means 算法的基本过程如下：①首先从 n 个数据对象中选定 K 个不同的点作为初始质心点，每个质心可以看作是一个类别的标识点；②然后将数据集中每个点划分到距离最近的一个质心所对应的类别；③完成

一次类的划分后根据此次聚类的结果重新计算各个类别的新质心；④如果新的质心和之前的质心距离大于某个阈值，那么说明现在的聚类结果还没有达到最佳结果，需要进行下一次迭代，再根据新的质心对点进行分类，直到所有新的质心和之前的质心距离小于某个阈值，说明质心点基本不再更新，算法结束。

我们用一个简单的三维坐标点实例来说明如何使用 Spark 实现 K-means 算法。我们的原始数据内容如下。

input.csv

```
0.0   0.0   0.0
0.1   0.1   0.1
0.2   0.2   0.2
9.0   9.0   9.0
9.1   9.1   9.1
9.2   9.2   9.2
```

数据一共包括 6 个点的坐标，每一个点的坐标值用空格分隔。由原始数据我们可以很明显地看出前 3 个点应划分为一类，而后 3 个点应该划分为另一类。我们设定 K-means 算法希望聚类的个数 k 为 2。我们最终希望得到的结果为前 3 个点聚类后的质心点和后 3 个点聚类后的质心点，即如下的两个质心点坐标值。

```
9.1,  9.1,  9.1
0.1,  0.1,  0.1
```

为此，我们编写了如下代码以实现 K-means 算法。

K-means 算法代码

```
1:   scala> val k = 2
     k: Int = 2
2:   scala> val e = 0.1
     e: Double = 0.1
3:   scala> val maxIterations = 5
     maxIterations: Int = 5
4:   scala> var iteration = 0
     iteration: Int = 0
5:   scala> val data = sc.textFile("input.csv").map(x => x.split(" ").map(_.
toDouble))
     data: org.apache.spark.rdd.RDD[Array[Double]] = MapPartitionsRDD[2]
at map at <console>:21
6:   scala> data.cache
```

```
     res0: data.type = MapPartitionsRDD[2]at map at <console>:22
7:   scala> var centers:Array[Array[Double]] = _
     centers: Array[Array[Double]] = null
8:   scala> do{
         | centers = data.takeSample(true,2,System.nanoTime.toInt)
         | }while(centers.map(_.deep).toSet.size != k)
9:   scala> def euclideanDistance(xs:Array[Double],ys:Array[Double]) =
{ Math.sqrt((xs zip ys).map {
         | case (x,y) => Math.pow(y - x, 2) }.sum) }
        euclideanDistance: (xs: Array[Double], ys: Array[Double])Double
10:  scala> var changed = true
        changed: Boolean = true
11:  scala> val dims = centers(0).length
        dims: Int = 3
12:  scala> while(changed&& iteration <maxIterations){
         | iteration += 1
         | changed = false
         | //计算每个点距离最近的质心
         | val pointWithClass = data.map({ point =>
         | val closestCenterIndex = centers.zipWithIndex.map({case (center,
index) =>{
         | val distance = euclideanDistance(point,center)
         | (distance,index)
         | }}).reduce((d1,d2) => if(d1._1 > d2._1) d2 else d1)._2
         | (closestCenterIndex,(point,1))
         | })
         | //计算每个新质心所属点的坐标总和还有点的个数
         | val totalContribs = pointWithClass.reduceByKey({ case ( (xs,c1),
(ys,c2) ) =>
         | ( (xs zip ys).map{case (x,y) => x + y},c1 + c2)}).collect
         | //计算新的质心
         |    val newCenters = totalContribs.map{
         |    case (centerIndex,(sum,counts)) =>
         |    (centerIndex,sum.map(_/counts))}.sortBy(_._1).map(_._2)
         |    for(i <- 0 until k){
         |        if(euclideanDistance(centers(i),newCenters(i))>e) {
         |            changed = true;
         |            centers(i) = newCenters(i)
         |        }
         |    }
         | }
13:  scala> centers.foreach(x => println(x.mkString(",")))
    9.1,  9.1,  9.1
    0.1,  0.1,  0.1
```

代码的执行过程如图 5-7 所示。代码的 1 ~ 4 行定义了算法的配置参数包括：①希望将数据分为 2 类；②算法的终止条件为，如果新质心与原质心的距离小于 0. 1

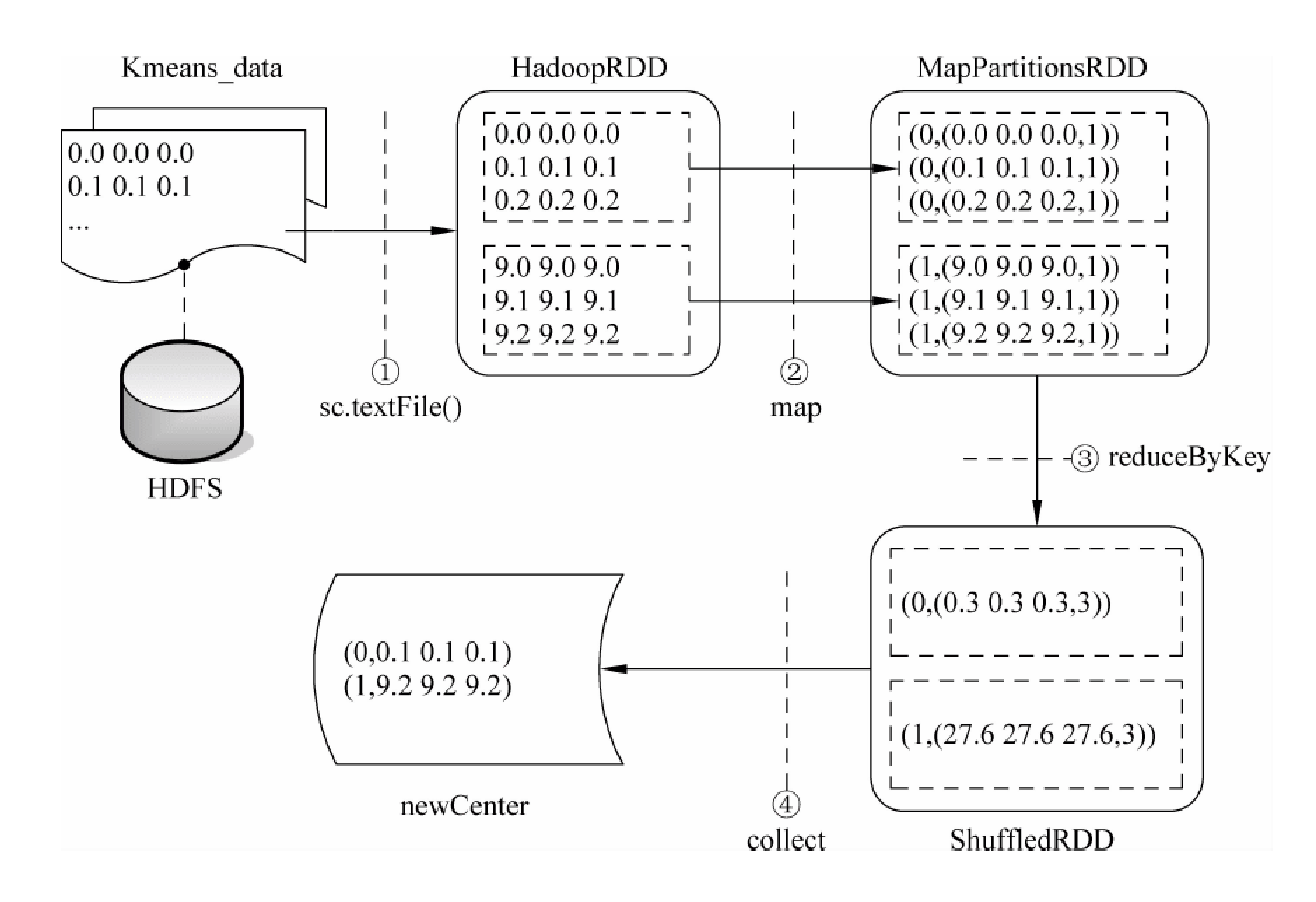


图 5-7 K-means 算法过程

或者迭代次数大于 5 次；③算法的起始迭代从第 0 次开始。代码第 5 行使用 sparkContext 类的 textFile 方法读取输入文件，并将每一行的坐标值用空格分隔，最后将坐标值转化为 Double 数字类型以便后续的距离计算。因为 K-means 算法是一个反复迭代的算法，需要重复使用各个点的坐标值来计算新的质心，因此代码第 6 行将点的坐标值使用 cache 存入内存中来加快运算速度。代码第 7 行定义了一个用来存储质心点的数据集合。代码第 8 行随机选取输入数据中的两个点作为初始质心，存入代码第 7 行定义的集合中。由于是随机选取的点，因此我们使用了一个 do-while 循环来保证我们取到两个点为不同的点。代码第 9 行自定义了一个计算两个点欧式距离的函数，在函数中使用 scala 的 zip 函数来保证两个点对应位置坐标值的计算。代码第 10 行定义了一个循环的判断标志。代码第 11 行得到了点坐标的维数。代码第 12 行是算法的核心部分，算法是一个循环更新的过程，循环体内部总共包括 3 个部分。第一部分计算得到 pointWithClass，计算出每个点所属的类别（也就是质心），pointWithClass 中存储了各个点所属的质心点、点的坐标值以及个数标识值 1。在计算 pointWithClass 时首先使用 map 算子计算每一个点到不同质心的距离，再使用 reduce 算子得到距离最近的质心点，将距离最近的作为该点属的质心点输出。第二部分计算得到 totalContribs，totalContribs 中存储了属于某个质心点的所有点对应坐标值的总和以及属于该质心的所有点的个数，计算时使用 reduceByKey 将所有点依据所属的质心进行分类，在每个分类内计算所有点的对应坐标值的总和以及属于该类的点的个数总和。然后计算得到

newCenters，newCenters存储了新的质心点，计算newCenters时先使用map算子计算每一个新的质心点，再判断每一个新的质心点与原质心点的距离是否大于初始设置的距离，如果距离大于初始设置的距离则进行更新循环继续进行；如果距离全部小于初始设置的距离则循环计算结束。最后代码第13行输出打印所有质心点。

5.8 关联规则挖掘

如今大型的综合性网络购物平台越来越多，像国内的淘宝、天猫、京东、当当等，国外的亚马逊、ebay等。这些互联网购物平台使得人们足不出户就可以买到商品，但由于互联网购物平台不用真实的卖场展示各种商品，同时每个互联网购物平台都经营着种类繁多的商品，这就使得顾客想从浩瀚如海的商品中找到适合自己的商品非常困难。为此，各个互联网购物平台无一例外地都会通过推荐系统来根据每个客户的兴趣特点和以往的购买行为，向用户推荐可能希望购买的商品。推荐系统往往是一个庞大的系统，算法也多种多样，这里我们不去逐一介绍，只是通过其中一种简单有效算法——关联规则挖掘[22]来展示Spark的算法设计。

关联规则挖掘就是在大量数据集中发现隐藏的有意义的联系，将这些联系用关联规则来描述，即计算在一些事件发生的情况下，另一个事件发生的可能性。一条关联规则可以描述为$X \rightarrow Y$，其中，X和Y都是一个条件项集，X称为规则的左部，Y称为规则的右部，并且满足条件$X \cap Y = \Phi$，这条规则表示X出现时Y也会出现。关联规则的强度，即它有多大概率是成立或可用的，由支持度（$s(X \rightarrow Y)$）和置信度（$c(X \rightarrow Y)$）来描述，它们的定义如下。

$$s(X \rightarrow Y) = \frac{|\{t_i | X \subseteq t_i, Y \subseteq t_i, t_i \in T\}|}{|\{t_i | t_i \in T\}|}$$

$$c(X \rightarrow Y) = \frac{|\{t_i | X \subseteq t_i, Y \subseteq t_i, t_i \in T\}|}{|\{t_i | X \subseteq t_i, t_i \in T\}|}$$

其中，符号$|\cdot|$表示集合的元素总数，T是事务总集合，ti是事务。可以看到，支持度即包含X和Y的事务在总事务中的占比，它描述了支持该关联规则的事务基数是否足够大。如果支持度太低，说明即使在过往的记录中X和Y一定会被一起购买，但这可能只是偶然出现，并没有什么意义。置信度即在一个事务中当X出现时Y出现的概率，它描述了Y伴随X出现的可能性大小，置信度越高，说明通过该规则进行推理越可靠。

在这里我们用一个简单的实例来说明如何使用Spark实现关联规则挖掘算法。我们的原始数据是一些用户购买商品的记录，其内容如下。

```
inputFile.csv
```

```
牛奶,咖啡,牛肉
牛奶,火腿,牛肉
猪肉,牛肉
牛肉,猪肉
火腿,牛肉,牛奶
```

原始数据中的每一行被称作一个事务,每个事务包含顾客一次购买的全部商品。事务中的每一种商品被称为一个项,如“牛奶”是一个项、“火腿”也是一个项。包含 0 个或多个项的集合被称为项集。因此每一个事务可以看作是一个项集,事务的一个子集也可以看作是一个项集。如果一个项集包含 k 个项,则称它为 k-项集。对于一个给定的事务集合 T,它包含的项集可以分为 1-项集,2-项集,……,k-项集,k 是 T 中的最大事务宽度(即事务所包含的项数)。例如,对于上面的输入事务集合 T,最大的事务宽度为 3,因此可以生成 1、2、3 项集。如表 5-3 所示。

表 5-3 项集规则实例

项 集	关 联 规 则
1-项集	{牛奶}、{咖啡}、{牛肉}、{火腿}、{猪肉}
2-项集	{牛奶,牛肉}、{咖啡,牛奶}、{咖啡,牛肉} {火腿,牛奶}、{火腿,牛肉}、{牛肉,猪肉}
3-项集	{咖啡,牛奶,牛肉}、{火腿,牛奶,牛肉}

其中每个项集内可以生成若干候选关联规则,如 3-项集{咖啡,牛奶,牛肉}可以生成以下的候选关联规则。

{牛奶,咖啡}→{牛奶}　　{牛奶,牛肉}→{咖啡}　　{咖啡,牛肉}→{牛奶}

{牛奶}→{咖啡,牛肉}　　{咖啡}→{牛奶,牛肉}　　{牛肉}→{咖啡,牛奶}

而这些候选关联规则是否最终能够成为可用的关联规则就要计算规则的支持度和置信度了。这里我们设置支持度大于 0.4 和置信度大于 0.7 的规则才最终确定为可用的规则。另外,为了提高算法的速度,通常会先舍弃支持度不够的关联规则,不再对它们计算置信度。从支持度的定义可以知道,它只跟项集 $X \cup Y$ 在事务集合中出现的频数有关,即同一个项集 $X \cup Y$ 生成的所有关联规则的支持度是一样的。因此,由一个项集生成的所有关联规则的支持度,可直接由这个项集出现的频数和总事务数计算得到。为了更清楚地说明关联规则挖掘算法,我们只考虑简单的右部集为 1-项集的关联规则,即希望得到如下规则结果。

outputFile.csv

```
(List(猪肉),List(牛肉),1.0,0.4)
(List(火腿,牛奶),List(牛肉),1.0,0.4)
(List(牛奶),List(牛肉),1.0,0.6)
(List(火腿,牛肉),List(牛奶),1.0,0.4)
(List(火腿),List(牛奶),1.0,0.4)
(List(火腿),List(牛肉),1.0,0.4)
```

为此，我们设计了以下代码完成以上规则挖掘工作。

关联规则挖掘算法代码

```
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.SparkContext._
object AssociationRuleMiningDemo {
    def main(args: Array[String]): Unit = {
        if (args.length < 2) {
            System.err.println("Usage: AssociationRuleMiningDemo <transactions> <output path>")
            System.exit(1)
        }
        val Array(transactionsFileName, outputPath) = args
        val sparkConf = new SparkConf().setAppName("Association Rule Mining Demo")
        val sc = new SparkContext(sparkConf)
        val transactions = sc.textFile(transactionsFileName, 2).cache
        val transactionSize = transactions.count();
        val itemsets = transactions.map(toList).map(findSortedCombinations(_)).flatMap(x => x).filter(_.size > 0).map(x => (x, 1L)).cache
        val minSup = 0.4
        val combined = itemsets.reduceByKey(_ + _)
            .map(x => (x._1, (x._2, x._2.toDouble / transactionSize.toDouble)))
            .filter(_._2._2 >= minSup).cache
        val subitemsets = combined.flatMap(itemset => {
            val list = itemset._1
            val frequency = itemset._2._1
            val support = itemset._2._2
            var result = List((list, (List(""), (frequency, support))))
            if (list.size == 1) {
                result
            } else {
                for (i <- 0 until list.size) {
```

```
                    val listX = removeOneItem(list, i)
                    val listY = list.diff(listX)
                        result ++= List((listX, (listY, (frequency, support))))
                }
                result
            }
        }).cache
        val rules = subitemsets.groupByKey()
        val assocRules = rules.map(in => {
            val listX = in._1
            val listYLists = in._2.toList
            val countX = listYLists.filter(_._1(0) == "")(0)
            val newListYLists = listYLists.diff(List(countX))
            if (newListYLists.isEmpty) {
                val result = List((List(""), List(""), 0.0D, 0.0D))
                result
            } else {
                val result = newListYLists.map(t2 => (listX, t2._1, t2._2._1.toDouble
                 /countX._2._1.toDouble, t2._2._2))
                result
            }
        })
        val minConf = 0.7
        val finalResult = assocRules.flatMap(x => x).filter(_._3 >= minConf)
        finalResult.saveAsTextFile(outputPath)
        System.exit(0)
    }
    def toList(transaction: String): List[String] = {
        val list = transaction.trim().split(",").toList
        list
    }
    def removeOneItem(list: List[String], i: Int): List[String] = {
        if ((list == null) || list.isEmpty) {
            return list
        }
        if ((i < 0) || (i > (list.size - 1))) {
            return list
        }
        val cloned = list.take(i) ++ list.drop(i + 1)
        cloned
    }
    def findSortedCombinations[T](elements: List[T])(implicit B:
```

```
Ordering[T]): List[List[T]] = {
67:          val result = elements. sorted (B). toSet [T]. subsets. map (_.
toList).toList
68:          result
69:      }
70:  }
```

代码的执行过程如图 5-8 所示。代码第 12 行使用 sparkContext 类的 textFile 函数读取用户的交易数据文件 inputFile. csv 生成 ParallelizedRDD。代码第 13 行统计事务的总数,即交易数据文件 inputFile. csv 中的交易记录总数。代码第 14 行将每一行记录的购买商品进行分割,然后求出每一行事务的所有项集,最后将一行事务的所有项集以 Key/ Value 对的形式输出,其中 Key 是项集,Value 设为 1,最终将计算得到的结果存为 itemsets。itemsets 的计算使用了多个算子,首先调用一个 map 算子,该 map 算子调用 52 ~55 行定义的 toList 方法,toList 方法负责将每一行的各个购买记录以逗号进行分割,将分割后的所有项存入一个集合中。之后调用了另一个 map 算子,该 map 算子调用了 66 ~69 行定义的 findSortedCombinations 方法,findSortedCombinations 方法负责计算出集合的所有子集。之后调用 flatMap 算子将所有子集扁平化为独立的单元,

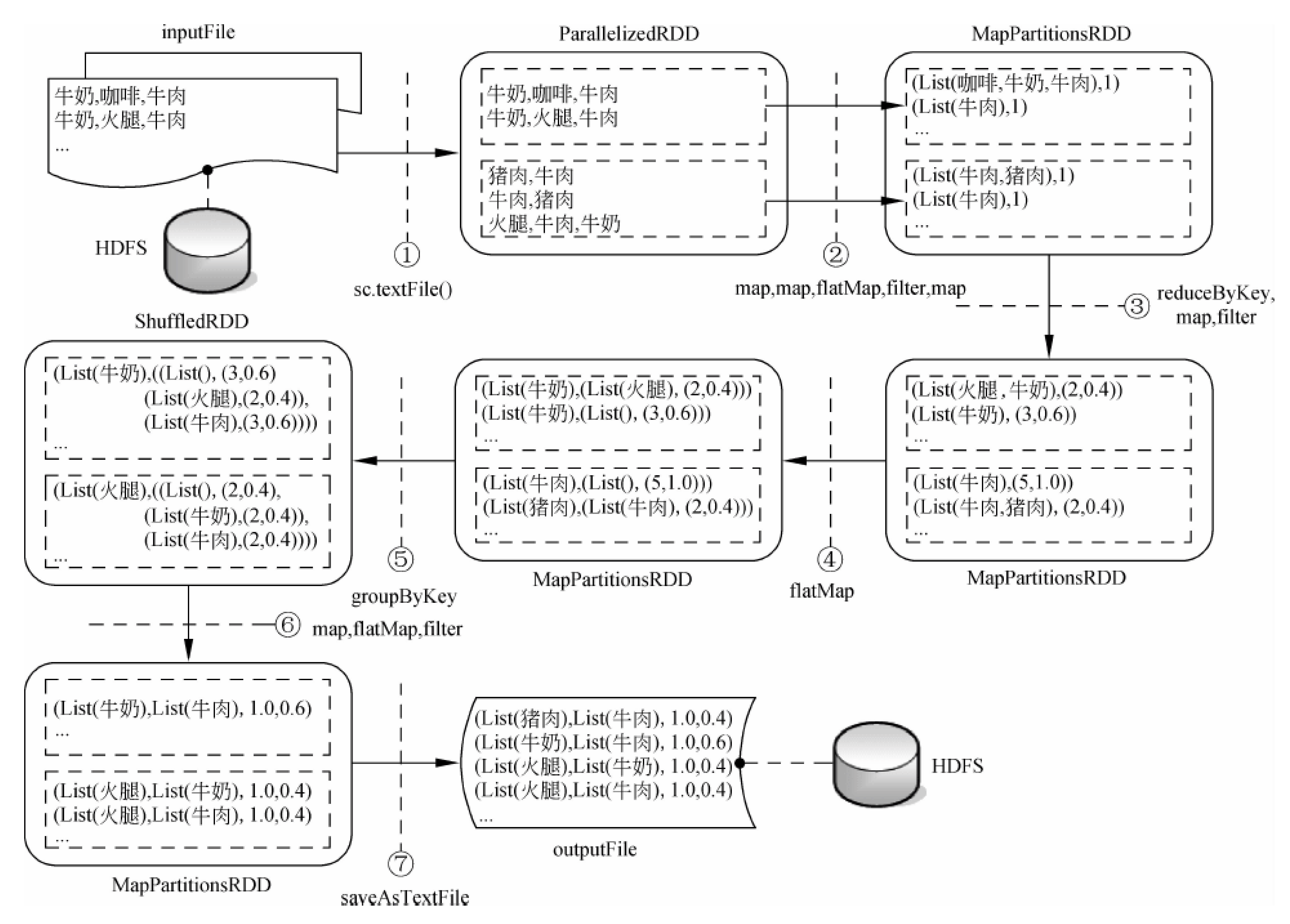


图 5-8　关联规则挖掘算法过程

再使用 filter 算子过滤掉大小为 0 的单元子集。最后使用 map 算子将单元子集以 Key/Value 对的形式存储成 itemsets。这里的单元子集就是上面介绍关联规则挖掘时提到的项集,以下将称其为项集,以对应上面的概念定义。代码第 15 行设置了支持度的阈值。代码第 16 行首先计算每个项集的支持度,然后过滤掉支持度没有达到阈值的项集,最终将结果存为 combined。计算 combined 同样使用了多个算子,首先使用 reduceByKey 计算每个项集在输入事务中出现的总次数,然后使用 map 算子计算每个项集的支持度,map 的输出包括 3 个值,分别是项集、项集的出现次数、项集的支持度,最后使用 filter 算子过滤掉小于阈值的项集。代码 17 ~ 32 行对支持度大于阈值的每一个项集进行处理,一个项集生成一个关联规则列表(只包含右部为 1-项集的关联规则),并使用 flatMap 算子将所有关联规则列表扁平化成一个个关联规则。在 flatMap 算子的输入函数中,代码首先以项集本身作为规则的左部、以空集作为规则的右部来生成一个规则;因为每个项集的出现总次数会在计算置信度时用到,因此对每个项集我们都生成了一个左部为项集本身的规则,用来表示这个项集的出现总次数。接着代码判断项集的大小是否为 1,如果项集的大小为 1,则说明该项集只包含一个商品,该项集不会生成其他规则;如果项集的大小不为 1,我们会遍历项集中的每一个商品,然后调用代码 56 ~ 65 行将该商品作为规则的右部,而将项集中的其他所有商品作为规则的左部,从而生成一条关联规则;最后返回的 result 即为包含该项集所生成的所有关联规则的一个列表;所有输入项集生成的多个列表由 flatMap 算子扁平化后,变成一条条关联规则。代码 17 ~ 32 行的最终输出同样是一个 key-value 型的 RDD,key 是规则的左部,value 包含 3 个部分,第一部分是规则的右部,第二部分是项集的出现次数,第三部分是规则的支持度。代码第 33 行使用 groupByKey 算子将 key 相同的规则聚为一组,即左部相同的规则会聚为一组,左部相同的规则包括两类:一类是 value 中的规则右部不为空的集合,标识了一个候选的规则,其 value 中的次数是该规则的出现次数;另一类是规则右部为空的集合,该 value 中的次数标识了规则左部项集的出现次数。之后代码 34 ~ 46 行会使用 map 算子在同一组中根据规则的出现次数和规则左部项集的出现次数来计算置信度。代码 47 行设置置信度的阈值。最后代码第 48 行使用 flatMap 和 filter 算子生成规则并过滤掉置信度没有达到置信度阈值的规则。代码第 49 行将结果存储输出。

5.9 kNN

kNN 算法[23]是数据挖掘算法中一个十分经典的分类算法,与之前我们介绍的 K-means 算法类似,kNN 算法也是基于样本点间距离计算的算法。但与 K-means 聚类算法不同,kNN 算法是一个分类算法。kNN 分类算法的输入不同于 K-means 聚类算法,kNN 算法的输入包括两个数据集,一个是样本点已有分类结果的数据集,另一个是样本点不知道分类结果需要通过 kNN 算法进行判断的数据集,kNN 算法会根据已知分类数据集对未知分类结果数据集进行分类判断。因此,kNN 算法不需要像 K-means 算法那样经过一个反复迭代训练模型的过程。对于输入的已知分类标签样本点数据集和未知分类标签的样本点数据集,kNN 算法会根据已分类的样本点来判断未分类样本点的类别。kNN 算法过程如下:①计算未知分类样本点与已知分类样本点的距离(通常使用欧氏距离);②选取 k 个距离最短的样本点,这里的 k 是根据具体情况而设定的,即根据实际情况选取参与投票的样本点数;③最后按照少数服从多数的原则进行分类,即这 k 个样本点属于哪个类型的数量最多,则将该未知分类的样本点分类为这个数量最多的类。

我们用一个简单的实例来说明如何使用 Spark 实现 kNN 算法。算法原始数据的输入文件包括两个文件,其内容如下。

输入数据 1:

```
inputFile1.csv
```

```
a 1 2 3 4 4
b 4 4 3 2 1
c 3 3 3 2 4
d 0 0 1 1 -2
a 1 2 3 4 5
b 5 4 3 2 1
c 3 3 3 3 3
d -3 -3 -3 -3 -3
```

输入数据 2:

```
inputFile2.csv
```

```
2 3 1 2 4
```

```
8 4 3 7 1
7 2 5 3 2
 -2 -3 1 9 0
 -1 -1 -1 -1 -1
0 0 0 0 0
0 7 6 5 3
```

inputFile1.csv 是带有分类标签的数据集，inputFile1.csv 包含 6 列值，第一列为记录的类别，后面 5 列为记录的不同属性值。inputFile2.csv 是我们需要对其进行分类的数据集，包括 5 列属性值。我们最终希望得到的输出结果如下，即每一条记录都在第一列加入一个类别标签。

outputFile.csv

```
c 2 3 1 2 4
b 8 4 3 7 1
b 7 2 5 3 2
a -2 -3 1 9 0
d -1 -1 -1 -1 -1
d 0 0 0 0 0
a 0 7 6 5 3
```

为此，我们编写了以下的代码以实现 kNN 算法。

kNN 算法代码

```
1: scala> val trainSet = sc.textFile("inputFile1.csv").map(line => {
        | val datas = line.split(" ")
        | (datas(0), datas(1), datas(2), datas(3), datas(4), datas(5))})
   trainSet: org.apache.spark.rdd.RDD[(String, String, String, String, String, String)] = MapPartitionsRDD[2]at map at <console>:21
2: scala> val bcTrainSet = sc.broadcast(trainSet.collect())
   bcTrainSet: org.apache.spark.broadcast.Broadcast[Array[(String, String, String, String, String, String)]] = Broadcast(2)
3: scala> var bcK = sc.broadcast(3)
   bcK: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(3)
4: scala> val testSet = sc.textFile("inputFile2.csv")
   testSet: org.apache.spark.rdd.RDD[String] = inputFile2 MapPartitionsRDD[4]at textFile at <console>:21
5: scala> val resultSet = testSet.map(line => {
        | val datas = line.split(" ")
        | val x = datas(0).toDouble
        | val y = datas(1).toDouble
        | val z = datas(2).toDouble
```

```
        |val t = datas(3).toDouble
        |val s = datas(4).toDouble
        |val trainDatas = bcTrainSet.value
        |var set = Set[Tuple7[Double, Double, Double, Double, Double,
Double, String]]()
        |trainDatas.foreach(trainData => {
        |val tx = trainData._2.toDouble
        |val ty = trainData._3.toDouble
        |val tz = trainData._4.toDouble
        |val tt = trainData._5.toDouble
        |val ts = trainData._6.toDouble
        |val distance = Math.sqrt(Math.pow(x - tx, 2) + Math.pow(y - ty, 2)
 + Math.pow(z - tz, 2) + Math.pow(t - tt, 2) + Math.pow(s - ts, 2))
        |set += Tuple7(x, y, z, t, s, distance, trainData._1)
        |})
        |val list = set.toList
        |val sortList = list.sortBy(item => item._6)
        |sortList.foreach(item => println(item))
        |var categoryCountMap = Map[String, Int]()
        |val k = bcK.value
        |for (i <- 0 to (k - 1)) {
        |val category = sortList(i)._7
        |val count = categoryCountMap.getOrElse(category, 0) + 1
        |categoryCountMap += (category -> count)
        |}
        |var rCategory = ""
        |var maxCount = 0
        |categoryCountMap.foreach(item => {
        |if (item._2.toInt > maxCount) {
        |maxCount = item._2.toInt
        |rCategory = item._1
        |}
        |})
        |("Test sample",x, y, x, t, s, "Test result",rCategory)
        |})
    resultSet: org. apache. spark. rdd. RDD [(String, Double, Double,
Double, Double, Double, String, String)] = MapPartitionsRDD[5]at map at <
console>:33
6:  scala> resultSet.saveAsTextFile("outputFile")
```

代码的执行过程如图 5-9 所示。算法的整体设计思想是将需要分类的数据集 inputFile2 拆分成不同的数据子集,然后对在不同的计算节点上并行的不同数据子集进行分类。代码的第 1 行使用 textFile 读取输入文件 inputFile1. csv,然后调用 map 算子将每一条记录用空格分割,最终 map 算子将 inputFile1. csv 中的每一行差分成 6 列值存入 trainSet 中。代码第 2 行调用 broadcast 将 trainSet 广播到各个计算节点。

代码第 3 行设置 kNN 算法中的 k 值并将其广播到各个计算节点。代码第 4 行使用 textFile 读取输入文件 inputFile2. csv 存入 testSet 中。代码第 5 行对 testSet 使用 map 算子，对 testSet 中的每条记录进行分类判断，在 map 算子内首先会计算每一条记录与 trainSet 中各个记录的距离，然后将计算的距离进行排序，再根据设置的 k 值选取与该记录距离最小的前 k 个已知分类记录，最后统计这 k 个已知分类记录所属的类别数和，最后将该条记录分配给类别数最高的类。代码第 6 行将最终结果进行存储。

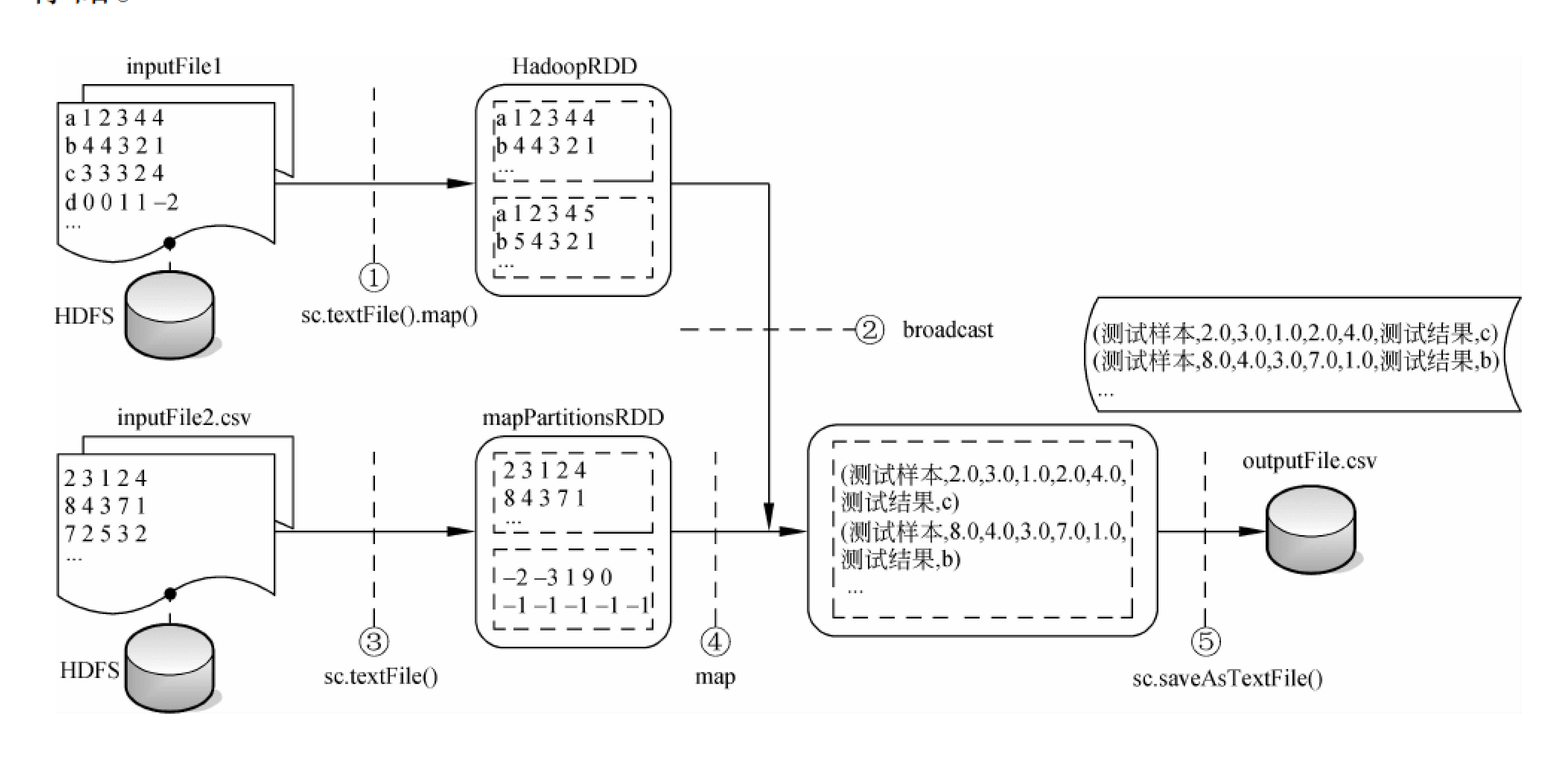


图 5-9 kNN 算法过程

5.10 朴素贝叶斯分类

在数据挖掘和机器学习领域，基于贝叶斯理论的数据分析方法占据了很重要的位置。其中一种非常简单但是很有效的分类算法就是朴素贝叶斯分类算法[24]。朴素贝叶斯分类算法属于线性分类方法，是基于贝叶斯定理的分类算法。贝叶斯定理的概率公式可用如下公式表示：

$$p(A \mid B)=\frac{p(B \mid A)p(A)}{p(B)} \tag{5-2}$$

公式中 $p(A)$ 是 A 的先验概率或边缘概率。之所以称为“先验”是因为它不考虑任何 B 方面的因素。$P(B|A)$ 是已知 A 发生后 B 的条件概率。$P(A|B)$ 是已知 B 发生后 A 的条件概率，被称作 A 的后验概率。$P(B)$ 是 B 的先验概率或边缘概率，也作标准化常量。贝叶斯定理的基本思想就是在已知先验概率 $p(A)$ 和条件概率 $P(B|A)$ 的基础上，对 A 发生的概率进行修正，得到后验概率 $P(A|B)$，最后再利用修正后

的概率 $P(A|B)$ 做出最优决策。在实际分类应用中，通常 A 代表分类值集合，B 代表属性值集合，各个分类值对应的 $P(B)$ 的概率都是一样的（反映了属性的本质自然规律），所以通常可以去掉 $P(B)$，即只需要计算 $P(B|A)P(A)$ 的大小即可。

“朴素贝叶斯”分类之所以被称为“朴素”的贝叶斯是因为该分类器在应用贝叶斯理论时，基于各属性之间具有独立性这个假设。我们知道，很多情况下，属性和属性之间是有关联的，而朴素贝叶斯方法直接忽略掉了这种关联性，使问题变得简单化，这也是其“朴素”的原因。简化后的分类值集合 A 中的每一个值 A_n 的朴素贝叶斯公式可以表示为：$P(A_n|B)=P(B_1|A_n)P(B_2|A_n)\cdots P(B_m|A_n)P(A_n)$。在公式中 $P(B_1|A_n)P(B_2|A_n)\cdots P(B_m|A_n)=P(B|A_n)$ 的转化正是应用了贝叶斯的“朴素”特性，将属性值 B 展开成各个独立属性值的条件概率相乘。

朴素贝叶斯分类器就是计算样本点 X 在具有 t 个属性 $(B_1,B_2,\cdots,B_t)$ 的情况下，对应于分类集合 $A=(A_1,A_2\cdots,A_n)$ 中所有值的朴素贝叶斯概率值，计算得到各个概率值 $p(A_1),p(A_2),\cdots,p(A_n)$ 后，比较各个概率值大小，将 $p(A_1),p(A_2),\cdots,p(A_n)$ 中的最大值作为样本点的分类值。尽管朴素贝叶斯分类器带着一些朴素思想和过于简单化的假设，但朴素贝叶斯分类器在很多复杂的现实情形中仍能够取得相当好的效果。

下面我们用一个简单的实例来说明如何使用 Spark 来进行朴素贝叶斯算法。算法原始数据的输入内容如下。

输入数据1：

```
inputFile1.csv
```

```
Sunny Hot High Weak No
Sunny Hot High Strong No
Overcast Hot High Weak Yes
Rain Mild High Weak Yes
Rain Cool Normal Weak Yes
Rain Cool Normal Strong No
Overcast Cool Normal Strong Yes
Sunny Mild High Weak No
Sunny Cool Normal Weak Yes
Rain Mild Normal Weak Yes
Sunny Mild Normal Strong Yes
Overcast Mild Normal Strong Yes
Overcast Hot Normal Weak Yes
Rain Mild High Strong No
```

输入数据 2：

inputFile2.csv

```
Overcast Mild Normal Weak
Sunny Mild Normal Weak
Rain Hot High Strong
```

输入包含两个文件，inputFile1.csv 中包含了外部环境天气情况，其特征属性值有天气、温度、湿度、风力 4 列。inputFile1.csv 中最后一列是某人是否出行的记录。我们需要根据 inputFile1.csv 训练一个判断某人是否出行的朴素贝叶斯分类器。当给定 inputFile2.csv 时，根据 inputFile2.csv 中给定的外部环境天气情况判断某人是否出行。我们最终希望得到的输出结果如下。

outputFile.csv

```
Overcast Mild Normal Weak Yes
Sunny Mild Normal Weak Yes
Rain Hot High Strong No
```

为此，我们编写了以下的代码以实现朴素贝叶斯分类。

朴素贝叶斯分类器代码

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
import scala.collection.mutable.{ArrayBuffer, HashMap}
object NaiveBeyes {
    def main(args:Array[String]) {
        val conf = new SparkConf().setAppName("Naive Bayes")
        val sc = new SparkContext(conf)
        val file = sc.textFile("inputFile1.csv", 2)
        val pair = file.flatMap(line =>{
            val tokens:Array[String] = line.split(" ")
            val classificationIndex = tokens.length - 1
            val theClassification:String = tokens(classificationIndex)
            var result =new Array[(String,String)](classificationIndex +1)
            var i =0
            while(i < classificationIndex){
                result(i) = ((tokens(i),theClassification))
                i +=1
            }
            result(i) = ("CLASS",theClassification)
```

```
            result
        }
        ).map(pair => (pair, 1))
        val counts = pair.reduceByKey(_+_)
        val countsAsMap = counts.collectAsMap()
        var CLASSIFICATION = new ArrayBuffer[String]()
        var PT = new HashMap[(String,String),Double]
        for((key,value)<-countsAsMap){
            val classification = key._2
            if(key._1 == "CLASS"){
                CLASSIFICATION += classification
                PT(key) = value
            }
            else{
              val sum = countsAsMap(("CLASS",key._2))
              if(value == null){
                  PT(key) = 0.0
              }
              else{
                PT(key) = value.toDouble/sum.toDouble
              }
          }
      }
      var trainingSize = 0.0
      for(classification <- CLASSIFICATION){
          trainingSize += PT(("CLASS",classification))
      }
      for(classification <- CLASSIFICATION){
          PT(("CLASS",classification))/ = trainingSize
      }
      val PT2save = PT.toArray
      val ptRDD = sc.parallelize(PT2save, 2)
      ptRDD.saveAsTextFile("PT")
      val CLFRDD = sc.parallelize(CLASSIFICATION, 1)
      CLFRDD.saveAsTextFile("CLASSIFICATION")
      val testdata = sc.textFile("inputFile2.csv", 1)
      val broadcastPT = sc.broadcast(PT)
      val broadcastCLASS = sc.broadcast(CLASSIFICATION)
      val classified = testdata.map(line =>{
        val attributes:Array[String] = line.split(" ")
        val PT = broadcastPT.value
        val CLASS = broadcastCLASS.value
        var selectedCLASS:String = null
        var maxProbility:Double = 0
        for(aCLASS <- CLASS){
            var postprob:Double = PT(("CLASS",aCLASS))
            for(i <- 0 until attributes.length){
                var prob:Double = 0.0
                if(PT.contains((attributes(i),aCLASS))){
```

```
69:                 prob = PT((attributes(i),aCLASS))
70:                 println("P(" + attributes(i) + "|" + aCLASS + ") " + prob)
71:               }
72:             postprob * = prob
73:           }
74:           if(selectedCLASS == null){
75:             selectedCLASS = aCLASS
76:             maxProbility = postprob
77:           }
78:           if(postprob > maxProbility){
79:             selectedCLASS = aCLASS
80:             maxProbility = postprob
81:             }
82:         }
83:         (line,selectedCLASS)
84:     })
85:     classified.map(x => x._1 + " " + x._2)saveAsTextFile("outputFile")
86:   }
87: }
```

代码的执行过程如图 5-10 所示。代码第 8 行使用 sparkContext 类的 textFile 函数读取 inputFile1.csv 文件生成 HadoopRDD。接下来算法需要计算 inputFile1.csv 文件中每个属性在类别“是否出行”下的条件概率 $P(B_1|A_n)P(B_2|A_n)\cdots P(B_m|A_n)$，($B_m$ = 天气、温度、湿度、风力)以及每个类别的先验概率 $P(A_n)$(A_n = yes、no)。$P(B_m|A_n)$等于(B_m,A_n)出现次数除以 A_n的出现次数。而 $P(A_n)$等于 A_n的出现次数除以实验样本总数 $\sum A_n$ (A_n = yes、no)。因此算法需要统计(B_m,A_n)出现次数以及 A_n的出现次数。计算(B_m,A_n)出现次数以及 A_n的出现次数的过程与经典的 WordCount 程序十分类似，一共分为两步。第一步，我们将每一条的样本记录进行拆分，拆分成属性与类别的组合对即(B_m,A_n)，代码 9 ~ 21 行使用 flatMap 算子完成了上述功能，在 flatMap 算子内部首先将每行样本记录以空格分割，然后将每行前 4 列的属性值分别与第 5 列的类别值进行组合，生成属性与类别的组合对。第二步，为了统计类别 A_n的出现次数，代码 19 ~ 20 行对每行类别列做了特殊处理，将每行的类别和字符串“CLASS”组合成(CLASS,A_n)对，其目的就是在后续的程序里方便对类别的出现次数求和。flatMap 算子最后会将每一个属性与类别的组合对扁平化为独立的单元。和 WordCount 程序一样，代码第 22 行将 flatMap 输出的每一独立单元都生成一个 pairRDD。其中 Key 为属性与类别组合对(或者是“CLASS”与类别组合对)，Value 值为 1，表示该组合对在样本记录中出现过一次。代码第 23 行使用 reduceByKey 算子统计每一个属性与类别组合对的次数以及每一类别的次数。代码第 24 行把上一步的统计结果执行 collectAsMap 算子，将统计

结果取回本地，存入 Hashmap 中（变量名为 countsAsMap）。因为接下来我们会用全部的统计结果来计算条件概率和先验概率，而 RDD 是分布式存储在不同计算节点上的，为了得到全部的统计值，需要把数据取回本地。代码 27 ~ 43 行算法在本地计算全部的条件概率。代码 43 ~ 49 行算法在本地计算全部的先验概率值。代码 50 ~ 52 行使用 parallelize 将先验概率值重新转化为 RDD 并使用 saveAsTextFile 将先验概率存储到 HDFS 中。代码 53 ~ 54 行同样使用 parallelize 和 saveAsTextFile 将条件概率转化为 RDD 存储到 HDFS 中。

到这里，我们的朴素贝叶斯分类器已经训练完成，接下来就是用我们训练的分类器来预测新数据了。代码第 55 行使用 textFile 读入另一个需要分类的输入数据 inputFile2. csv。代码 56 ~ 57 行将 HDFS 上存储的先验概率文件和条件概率文件作为广播数据分发到各个计算节点。代码 58 ~ 84 行使用 map 算子计算输入数据 inputFile2. csv 中每一条记录的后验概率 $P(B_1|A_n)P(B_2|A_n)\cdots P(B_m|A_n)P(A_n)$，并根据概率值进行分类。最后在代码第 85 行将结果进行存储。

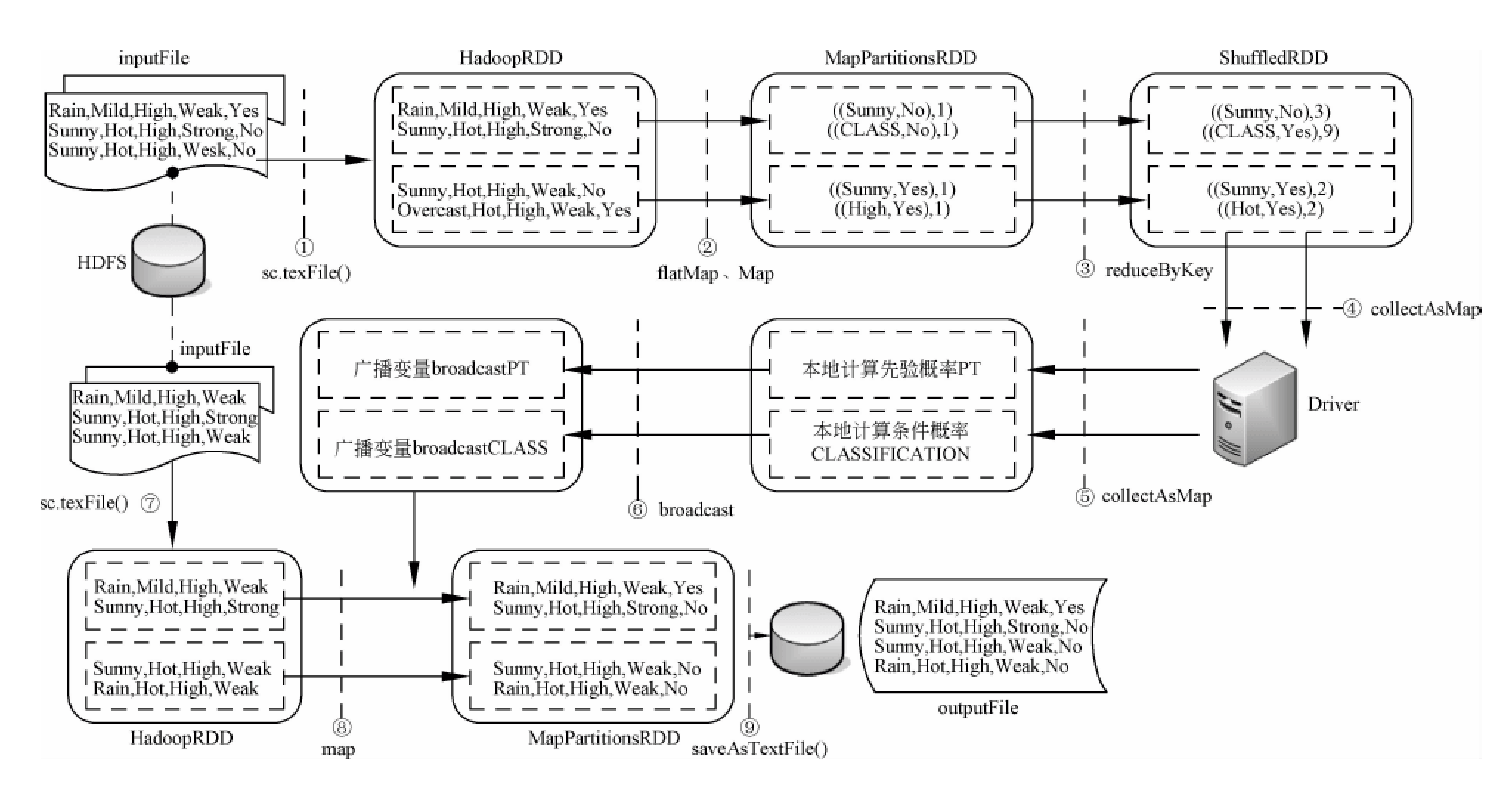


图 5-10 朴素贝叶斯算法过程

第6章

善用 Spark

至此,我们不仅已经了解了 Spark 的系统结构和运行原理,还通过大量的实例代码掌握了如何使用 Spark 脚本对数据进行简单的处理和编写 Spark 程序实现复杂的算法。我们利用简洁高效的 Scala 语言编写的程序,会由 Spark 执行引擎遵循 Job、Stage、Task 的顺序生成并行执行的任务在集群上自动运行。可以说,Spark 为我们完成了程序运行并屏蔽了实现分布式并行程序的底层细节。但是,由于分布式并行和集群运行环境的的复杂性,Spark 的默认策略并不一定能帮助我们实现算法和程序的最优化。因此,在这一章让我们一起深入 Spark 内部,探寻如何利用 Spark 的一些高级特性,使我们的算法和程序能更加优化和高效。需要说明的是,Spark 的 API 及生态系统都十分丰富和强大,很难在有限的篇幅中阐述太多的 Spark 使用技巧。在本章中,我们尽力将我们在实践过程中积累的重要环节和方面进行总结,为大家优化 Spark 程序提供帮助和启发。

6.1 合理分配资源

由于 Spark 程序运行在由多个服务器构成的集群中,需要通过外部的集群资源管理框架调用服务器资源。因此,掌握集群资源调度方式并在此基础上通过配置为程序合理分配资源,是优化 Spark 程序运行性能的首要任务。当前存在多种支持 Spark 程序运行的资源管理框架,例如 YARN、Mesos 等。这些系统针对 Spark 程序的资源分配有稍许差别,但在资源分配的优化思路上是一致的。在这里我们以 Spark on YARN 为例来进行后续的讨论。

在 YARN 集群中作业运行模式分为 yarn-client 和 yarn-cluster 两种模式,两者在资

源分配方式上差别不大,唯一的重要区别是 yarn-cluster 模式中主节点需要执行 Driver 进程。YARN 采用 Master/Slave 模式构建和管理整个架构,其中 Master 节点为 ResourceManager,它负责对整个集群的资源进行统一管理分配和作业运行调度。Slave 节点为 NodeManager,它负责管理单一节点上的资源并在该节点上启动 Container 运行任务。各个 NodeManager 会定期向 ResourceManager 汇报运行信息。在 Spark on YARN 运行方式中,生产环境一般使用 yarn-cluster 模式。该模式下,用户通过客户端提交一个 Spark 程序请求到 YARN 的 ResourceManager 后,ResourceManager 会为该应用在某个 NodeManager 中启动一个 Container 用于运行该程序对应的 Application Master。Application Master 主要负责与 ResourceManager 通信申请资源并驱动应用执行。ResourceManager 接收到 Application Master 的请求后为其分配 Container。然后 Container 会启动一个 Spark Executor 来执行后续一系列的 Task,该过程如图 6-1 所示。

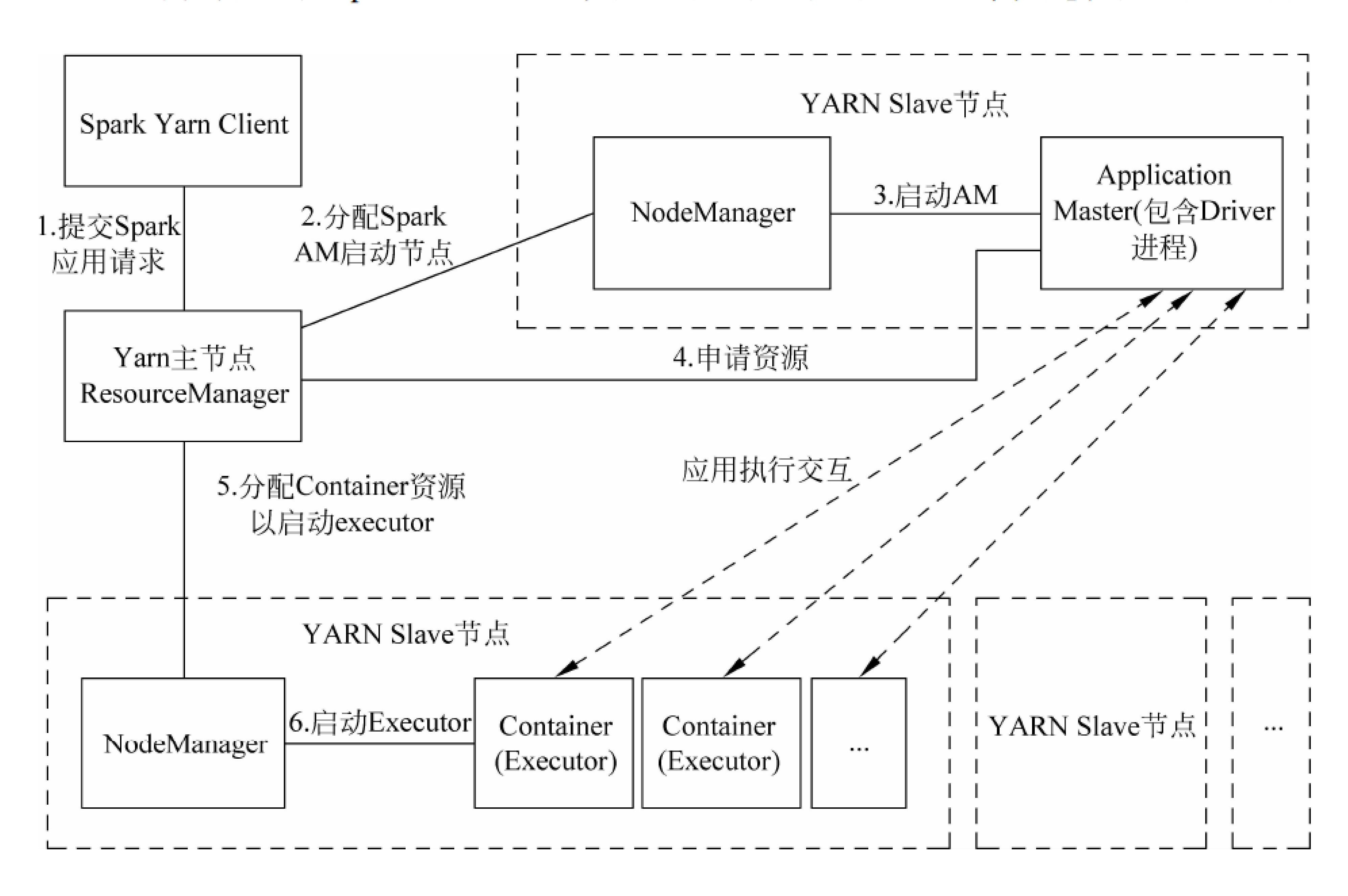


图 6-1 yarn-cluster 模式下的 Spark 程序执行流程

需要说明的是,Spark 程序运行需要调用的资源包括 CPU、内存、存储和网络,其中存储和网络这类资源与具体存储数据的文件系统、网络环境关系较为紧密,在 Spark 框架中通过参数对这两类资源进行优化的手段不多,因此在这里我们主要关注对 CPU 和内存资源的分配和优化。下面我们分别从 CPU 资源和内存资源两个方面了解一下优化手段。

√ **CPU 资源优化**

在对 Spark on YARN 工作方式下的 Spark 程序进行 CPU 配置优化时，主要是针对两个组件，Driver 和 Executor。针对 Driver 的优化比较简单，可以通过配置参数 driver-cores 为 Driver 分配合理的 CPU 核数，在配置的时候注意兼顾 Driver 和 Executor 的负载差异即可。

针对 Executor，我们可以通过以下方式优化配置。

- 在启动 Spark 程序时可以通过 Spark-shell 命令设置 num-executors 来指定启动的 Executor 数量，如果是 yarn-client 模式则还可以通过配置 spark-environment 中的 spark. executor. instances 配置参数来指定，默认 Executor 数为 2。
- 每个 Executor 使用的 CPU 核数可以通过 Spark-shell 命令设置 executor-cores 参数来指定，或者是通过配置 spark-environment 中的 spark. executor. cores 参数来指定，默认是 1 个核。
- 需要注意 YARN 中的参数 yarn. nodemanager. resource. cpu-vcores，该参数用于设置在某个 Slave 节点中所有 Container 可以使用的 CPU 核数。从 Spark1. 3 开始支持通过配置 Spark. dynamicAllocation. enabled 属性在 Spark on YARN 模式下动态分配 Executor。不过这个属性在 Spark 的 standalone 和 Mesos 模式下还没有作用。
- 当使用 HDFS 作为 Spark 程序的文件存储系统时，由于 HDFS 在存在大量客户端线程时，容易在高并发请求时出现性能问题，因此，为每个 Executor 配置的 CPU 核数建议不超过 5 个。当然，我们也不能将每个 Executor 的 CPU 核数设置太小，否则无法发挥 Executor 可以同时运行多个任务的优点，同时会影响广播变量使用时的性能。

√ **内存资源优化**

与 CPU 核数的配置数量只会影响 Spark 程序执行的快慢相比，由于 Spark 的计算模式极度依赖分布式内存对象，内存太小的配置不仅会影响性能，还有可能在配置不当时直接导致 Spark 程序执行失败。因此，在为处理大量数据的 Spark 程序进行配置时，尤其要重点关注内存的分配。在 Spark on YARN 模式下，与内存资源分配相关的参数、含义、使用方法和默认值如表 6-1 所示。

表 6-1 Spark on YARN 内存相关参数说明

属性	说　　明	配 置 方 法	默认值
Driver 内存	Driver 进程使用的内存大小（在 Spark on YARN 中 Driver 对应为 YARN 中的 Application Master）	配置参数 spark. driver. memory yarn-client 模式下命令参数——driver-memory	1GB
Driver 堆外内存	yarn-cluster 模式下 Driver 使用的堆外内存大小（Driver 内存 × 比例，比例因 Spark 版本不同而不同）	配置参数 spark. yarn. driver. memoryOverhead	最小为 384 MB
AM 内存	Application Master 使用的内存大小	配置参数 yarn. app. mapreduce. am. resource. mb	1536MB
AM 内存（client）	client 模式时，Application Master 的内存大小	配置参数 spark. yarn. am. memory	512 MB
AM 堆外内存	yarn-client 模式下 Application Master 使用的堆外内存大小（AM 内存 × 比例，比例因 Spark 版本不同而不同）	配置参数 spark. yarn. am. memoryOverhead	最小为 384 MB
Node Nanager 内存	每个 Nodemanager 可以使用的内存大小	配置参数 yarn. nodemanager. resource. memory-mb	8GB
Executor 内存	Executor 使用的内存大小	配置参数 spark. executor. memory 命令参数 executor-memory	1GB
Executor 堆外内存	Executor 使用的堆外内存大小（Executor 内存 × 比例，比例因 Spark 版本不同而不同）	配置参数 spark. yarn. executor. memoryOverhead	最小为 384 MB
Container 最小内存	一个 container 能够申请的最小内存大小	配置参数 yarn. scheduler. minimum-allocation-mb	1GB
Container 最大内存	一个 Container 能够申请的最大内存	配置参数 yarn. scheduler. maximum-allocation-mb	8GB
Container 内存增量	Container 申请资源增加的最小单位值	配置参数 yarn. scheduler. increment-allocation-mb	512MB（CDH）

这些参数中，Executor 分配的内存大小非常重要，其影响范围较为广泛，对包括 cache、group、join 在内的很多算子性能有很大的影响。我们也需要关注 YARN 里面的资源分配，如 yarn. nodemanager. resource. memory-mb 参数的值，它控制一个节点上所有 Container 可用的内存总和。同时，由于 JVM 在运行时会使用堆外内存，因此，堆外内存的大小也是一个需要关注的重要配置项。但是，如果为 JVM 配置过大的内存也可能导致过多的垃圾回收操作，从而影响程序性能，从我们的经验来看为每个 Executor 配置的内存大小不要超过 64GB。

在 Spark on YARN 的 Cluster 模式下，NodeManager 的内存属性层级如图 6-2 所示。在此模式下 Application Master 是一个不执行 Executor 的 Container，它需要运行 Spark 程序的 Driver 进程，因此需要为 Driver 分配相应的内存和 CPU，在 Client 模式下则不用。

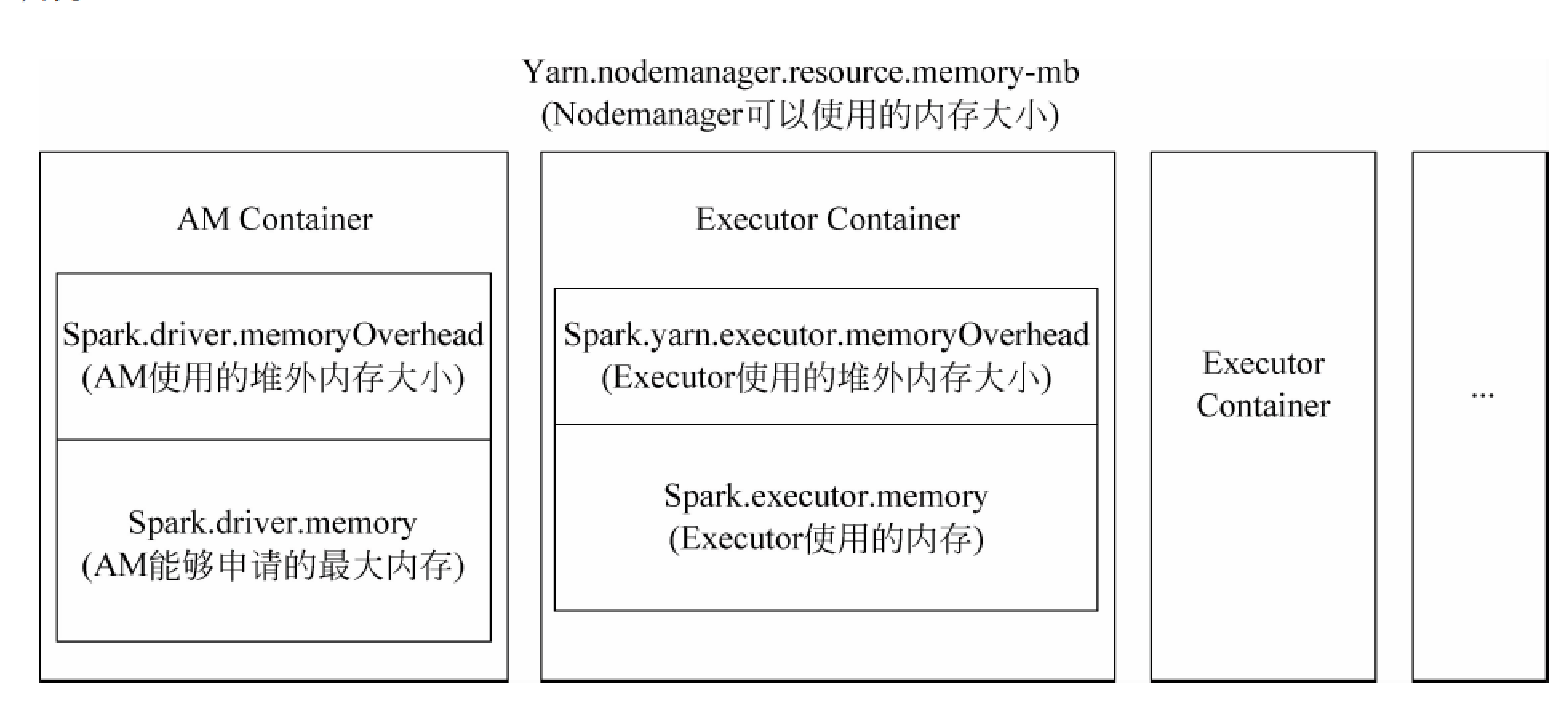


图 6-2 Spark on Yarn 在 Cluster 模式下的内存层级属性

在 Spark on YARN 的 Cluster 模式下，我们可以通过命令行参数 num-executors、executor-cores、executor-memory 和配置参数 spark. driver. memory 这 4 个参数的不同值的组合尝试设置最为合理的资源配置。我们构建了一个示例实验环境来说明这些参数不同配置情况下资源分配的情况。假设我们的集群中有 16 个节点，每个节点有 64GB 内存，2 颗 CPU，每颗 CPU 具有 16 个逻辑核。由于该集群要被多个项目组共享，因此我们需要根据 Spark 程序的不同类型，调整 CPU 和内存资源的配置比例来合理分配资源。例如，我们应为计算密集型程序配置较多的 CPU 资源，为数据密集型程序配置较多的内存资源。我们试验性地在该集群中配置了不同的参数值，并通过 YARN 和 Spark 的 Web 监控页面观察资源分配结果，如表 6-2 所示。

表 6-2 不同参数配置的资源分配情况

参数说明 \ 实验编号		1	2	3	4	5
Spark -submit 参数设置	Executor 数量 ——num-executors	8	8	8	8	4
	Executor 的 CPU 核数 ——executor-cores	4	4	4	8	4
	Executor 内存 ——executor-memory	4GB	4GB	8GB	4GB	4GB
	Driver(AM)内存 spark. driver. memory	2GB	4GB	2GB	2GB	2GB
实际运行效果	实际启动 Container 数	9	9	9	9	5
	Application Master 分配的内存	2560MB	4608MB	2560MB	2560MB	2560MB
	每个 Executor Container 分配的内存	4608MB	4608MB	9216MB	4608MB	4608MB
	YARN 实际分配内存	39424MB	41472MB	76288MB	39424MB	20992MB
	Executor 最大可用 Cache 内存	2. 1GB	2. 1GB	4. 1GB	2. 1GB	2. 1GB
	AM 最大可用 Cache	983. 1MB	1966. 1MB	983. 1MB	983. 1MB	983. 1MB

从表 6-2 的数据我们可以看到,集群最终的资源分配情况与我们期望的配置参数有一定区别,我们以实验 1 为例来解释一下这种区别及原因。

(1) 实际启动的 Container 数量为分配的 8 个 Executor 加上一个 Application Master 对应的 Container,因此为 9 个 Container。其中 Executor 对应的 Container 每个分配了 4 个 CPU 核数,运行 4 个任务。Application Master 的 Container 则为默认的 1 个 CPU 核。

(2) Application Master 分配的内存包括分配的内存加上堆外内存大小,堆外内存根据表 6-1 中计算公式计算为 384MB。但由于当前集群中容器内存增量单位为 512MB,因此总的分配内存为 2GB + 512MB = 2560MB。

(3) 和 Application Master 分配的内存原理一致,Executor Container 分配的内存为 4GB + 512MB = 4608MB。

(4) YARN 实际分配内存为 8 个 Executor Container 分配的内存与 Application Master 分配的内存之和。

(5) Executor 最大可用 cache 内存的大小可以从 Spark UI 中"Executor"标签下的页面读取,在该页面中"Memory Used"一列中右侧的数字即为 Executor 最大可用 Cache 内存。在这里,我们详细解释这个值的计算方法。我们在实验中对每个 Executor 分配了 4GB 内存和相应的堆外内存。但是在程序运行时,程序运行环境和其他相关的系统

组件会消耗掉一部分的内存,扣除这部分内存剩下的才是可用的内存。为了展示这一数据,我们启动一个占用4GB 内存的 Scala 运行环境并检查可用内存,如下面的代码所示。从结果中可以看出该环境中最大的可用内存为 4116709376 Byte,即 3926MB。

使用 Shell 命令检查可用内存

```
 $ scala - J - Xms4G - J - Xmx4G
welcome to scala vesion 2.10.4 (OpenJDK 64 - Bit Server VM, Java 1.7.0_45).
Type in expressions to have them evaluated.
Type :help for more information
scala > Runtime.getRuntime.maxMemory
res0: Long = 4116709376
```

另外,我们阅读 spark 源码中用于获取最大可用 Cache 内存的源码,该源码所在的文件为 spark/ core/ src/ main/ scala/ org/ apache/ spark/ storage/ BlockManager. scala,该段源码如下所示。

获取最大可用 cache 内存源码

```
private def getMaxMemory(conf: SparkConf): Long = {
  val memoryFraction = conf.getDouble("spark.sotrage.memory Fraction",
0.6)
  val safetyFraction = conf.getDouble("spark.storage.safety Fraction",
0.9)
  (Runtime.getRuntime.maxMemory * memoryFraction * safety Fraction).
toLong
}
```

我们可以计算出当前 Executor 最大可用 cache 内存为 3926MB ×0.6 ×0.9 = 2120.04MB,也就是我们从 spark UI 中看到的 2.1GB。需要说明的是,这个最大可用 Cache 内存并不是专门为 Cache 保留的内存,而只是 Cache 操作可以使用内存的一个上限。分配给 Executor 可以用的内存包括堆内存和堆外内存,最大可用的堆内存就是我们在 spark. executor. memory 配置参数和 executor. memory 命令行参数中设置的值(实验 1 中即为 4GB),而最大可用 cache 内存则是通过参数 executor. memory、spark. storage. memoryFraction(0.6)和 spark. storage. safetyFraction(0.9)算出来的 Cache 操作可以使用的最大堆内存,也就是被 cache 的 RDD 可以使用的堆内存。如果没有任何 RDD 被 cache,那么 Executor 可以任意使用堆内存直到其超过最大堆内存限制,即 executor. memory。如果有 RDD 被 cache 了,并且总大小超过了最大可用 Cache 内存,

那么被 cache 的 RDD 中最老的会被删掉，然后被新的替代。spark. storage. memoryFraction 参数就是为了防止那些被 cache 的 RDD 把堆内存耗光导致 Executor 无法创建新对象。Spark 里面还有另外一个参数是 spark. shuffle. memoryFraction（默认值为 0.2），是 Shuffle 的时候可以使用的最大堆内存，它跟 spark. storage. memoryFraction 是一个道理，用来限制 Shuffle 时可以使用的最大堆内存，默认情况下，即使 Shuffle 和 Cache 都用达到限制的最大值（0.8 × executor. memory），那么也还留有 0.2 × executor. memory 的堆内存空间给 Executor 创建新对象。

（6）AM 最大可用 Cache 内存同样取自 spark UI 的“Executor”标签页面“Memory Used”一列，它是 AM 中 Driver 进程最大可用 Cache 内存。它的计算方式和 Executor 最大可用内存计算方式是一样的，由于我们的应用一般不会把 RDD 在 Driver 上进行计算（只有一些很简单的操作，如 first、take 等，在设置了允许作业在本地运行的参数后，在 Driver 上才能计算 RDD），所以 Driver 最大可用 Cache 内存（实验 1 中即为 983.1M）实际上没什么意义，真正有意义的是在运行应用时指定的 driver-memory 参数，它才是 Driver 可以使用的最大堆内存。

综上所述，我们在为每个应用分配资源时，需要考虑自己应用对不同资源的需求比例进行合理的分配，防止资源浪费。同时需要在分配内存时对系统设置的堆外内存和最大可用内存进行充分考虑，防止分配的资源过小影响应用的执行。

6.2 控制并行度

我们都知道，Spark 程序是运行在由多台服务器构成的集群之上的分布式并行程序。显然，这样的程序在运行时的性能，一定会受到程序执行的并行度与集群可用资源是否匹配的影响。在探讨 Spark 程序并行度与集群资源关系之前，我们先来梳理下影响 Spark 程序在集群上运行性能的一些基本规则（这些规则我们在 Spark 原理章节部分已有介绍）。

√ Spark 中的核心数据结构 RDD 是由一系列的分区构成的，每个分区包含一个 RDD 中的部分数据；

√ 在 Spark 调度和执行计算任务（Task）时，会为 RDD 的每个分区创建一个 Task；

√ 默认情况下，Spark 会为每个需要执行的 Task 分配一个 CPU 内核资源。

基于以上规则，我们可以很直观地想到，在 Spark 上每个 Stage 执行的 Task 数量，与集群可用资源的匹配程度，直接影响着 Spark 程序的性能，这主要体现在以下方面。

√ 是否能充分利用集群资源决定了程序的运行时间：举例来说，如果我们拥有一个有 200 个 CPU 内核的集群，并且可以独占地被一个 Spark 程序使用。如果这个程序在某个 Stage 只启动了 20 个 Task，而每个 Task 都需要计算大量的数据，此时就意味着有大量的计算资源未能发挥作用。如果我们可以通过配置或算法优化提高该 Stage 启用的 Task 数量，则可以提高集群资源利用率，缩短程序运行时间。

√ 过小的并行度可能造成内存压力过大：Spark 的整体计算架构是围绕充分利用分布式内存对象而设计的。很多基础的算子，例如 join、cogroup 和 groupByKey 等常用算子，都需要在 HashMap 或内存缓存中保存大量的计算对象。同时，这些算子触发的 shuffle 操作在拉取数据时还需要使用大量的内存空间。此外，reduceByKey 和 aggregateByKey 等算子所触发的 shuffle 操作在拉取和输出时都需要占用大量的内存空间。在这样的情况下，如果程序的并行度太小，为这些算子所启动的每个 Task 将要处理规模很大的数据，集群内存如果不足以容纳这些数据，则可能出现严重的问题。此外，Spark 程序的运行是基于 JVM 的，如果内存空间不足会导致频繁或长时间的 GC 操作，可能会极大地增加程序的运行时间。同时，Spark 为了确保系统的健壮性，在并行度过小导致内存不足时会将数据临时写入磁盘中，将极大地增加磁盘 IO 时间及数据排序时间，同样会导致程序性能的恶化。

在默认情况下，Spark 执行引擎通过一些默认规则为我们解决了大部分程序并行度的控制工作，包括：

√ 在 Spark 中，RDD 间存在世系关系，在没有特别指定的情况下，每个 RDD 的分区数量通常是与其父 RDD 相同的。因此，对于没有 reduce 类算子的 Stage，Spark 执行引擎启动的 Task 数量通常与这个 Stage 的最后一个 RDD 的分区数相同。

√ 对于 reduce 类算子（例如 groupByKey 和 reduceByKey 算子）Spark 的基本规则是启动与父 RDD 中最大分区数相同的 Task 数量。

√ 在 Spark 里也存在要对没有父 RDD 的 RDD 启动 Stage 进行处理的情况，例如使用 textFile() 和 hadoopFile() 产生的 RDD。这些 RDD 的分区数是由文件大小及底层的 InputFormat 格式和参数决定的。通常情况下，这类 RDD 会对每个 HDFS 的文件块（Block）产生一个分区。

这些基本规则对于一些小规模的 Spark 程序，也能满足一些基本需求。但是这种默认的设置方式显然不一定是最优方案。例如，当 Stage 中的父 RDD 分区数很小，执行 reduceByKey 或 join 等需要执行分布式 Shuffle 变换时，启动的任务数也会很小，导致并行度不足。而使用 textFile( )基于 HDFS 文件产生的没有父 RDD 的 RDD 时，如果文件数量很多或文件很大，则可能因为文件块数量过大而启动过多的任务，增加了大量额外的通信开销。因此，我们也有必要来了解下 Spark 为我们提供了哪些手段可以控制程序的并行度，以使 Spark 程序与资源更好地匹配，包括：

√ 合理设置 spark. default. parallelism 参数：我们可以通过 spark-env. sh 脚本或在程序中修改 SparkContext. SparkConf 来设置 spark. default. parallelism 参数。该参数将控制 RDD 变换在没有设置变换后分区数时新生成的 RDD 的分区数量，同时将决定执行此变换时的任务数。

√ 在对 RDD 进行变换时利用参数控制分区数量：在前面介绍 RDD 算子时，我们已经看到在很多变换的函数接口中，均可以指定此变换生成的新 RDD 的分区数量，例如去重算子 distinct(numPartitions: Int)中的 numPartitions 参数。通过指定新 RDD 的分区数量，我们就可以控制后续计算时 Spark 引擎启动的任务数。虽然在 RDD 算子列表中我们没有列出所有具备指定分区数量参数的算子函数，但实际上大部分 RDD 算子是可以指定分区数量的，大家可以在使用时查阅 Spark 官方 API 文档。

√ 改变 RDD 的分区数量：Spark 为我们提供了一系列算子可以改变 RDD 的分区数量，从而达到控制程序并行度的目的。例如，repartition 算子可以将 RDD 进行 Shuffle 后生成指定数量的分区。如果我们明确地知道合并分区的方式，我们还可以使用 coalesce 算子而不进行 Shuffle 操作改变分区数量，由于减少了代价很高的 Shuffle 操作，coalesce 算子执行效率更高。

√ 控制 HDFS 的输入：用 HDFS 文件使用 textFile( )产生的 RDD 时，Spark 默认会为每个 HDFS 文件块创建一个分区。虽然在 textFile( )函数中我们可以指定希望的分区数量，但我们只能增加分区数而不能设置比 HDFS 文件块数更小的分区数。如果 HDFS 使用默认的 64MB(Hadoop 2. 0 之前)或 128MB(Hadoop 2. 0 之后)作为文件块大小，当我们觉得分区数过多时，可以通过增大 HDFS 文件块大小参数来达到减少分区数的目的。在 HDFS 文件块大小参数不能修改或 HDFS 文件已经存储完毕的情况下，则可以通过修改 Hadoop 的 mapreduce. input. fileinputformat. split. minsize 参数或编写自定义 InputFormat 类的方法来改

变读入数据的分区数量。

需要注意的是,太小的并行度不利于充分发挥集群的能力,而太大的并行度也可能会带来过多的网络通信开销降低性能。那到底多大的并行度是一个合适的值呢?这是一个很难回答的问题。由于要解决的问题和集群运行环境的复杂性,目前并没有一个统一的算法能帮助我们快速找到一个合适的并行度值。在实践过程中,我们发现 Sandy Ryza 在他的一篇博客中提到的一个实践性方法具有不错的效果(尽管他也提到了一个理论算法,但似乎很难应用)。在这里我们也分享一下参考这个实践方法的优化基本思路:首先通过 Spark UI 了解程序的 Stage 切分情况,然后针对每个 Stage 进行优化。对于每个 Stage 先从较小的并行度开始,不断通过 rdd. partitions( ). size( )算子关注关键 RDD 的分区数量,然后结合前面提到的 Spark 分区和任务数产生原则,以倍数提高该 Stage 的并行度,同时密切关注程序的运行时间。当并行度增加到运行时间不再缩短时,转而以 1.5 倍的比例从上一次并行度开始继续提高,直到运行时间不再缩短为止。

为了展示不同并行度对 Spark 程序的性能影响,我们设计了一个 HTTP 访问日志分析实例来进行说明。分析的输入文件是大量包含用户通过 HTTP 请求访问的 URL 的日志文件,存储在 logs 目录下。每个日志文件包含 67 个字段,包括时间、用户 ID 等,字段间采用“|”进行分隔。其中第 50 个字段为用户访问的 URL。文件格式和内容示例如下所示。

```
url01.csv
```

```
…abc.com/1.html …
…def.com.cn/2.php …
…
…uvw.com/3.jpg …
…xyz.com/4.png …
```

由于日志文件数据量非常大,为了节省数据分析消耗的时间和计算资源,我们并不希望对日志中的所有条目进行分析。例如,如果我们的目标是希望通过分析用户的点击行为来了解用户喜好,我们希望通过一些规则过滤掉非用户点击所产生的 URL,例如网页内嵌的图片等。我们将这些规则存储在文件 logs 目录下的 rule. csv 文件中。文件中每行为一个以正则表达式存储的规则。下面的代码展示了使用不同并行度对话单文件中全部记录进行正则匹配的性能比较。

不同并行度性能比较的示例程序

```
1: val file = sc.textFile("/logs/*", partitionNum)
2: val reg = sc.textFile("/logs/rule.csv").collect()
3: val broadReg = sc.broadcast(reg)
4: val validLogs = file.map(line => line.split("[|]")).filter(_.length ==
67).filter(!_(0).equals("Length")).
   filter(x => broadreg.value. map(x(50).matches(_)).reduce(_||_))
5: validLogs.map(_.mkString("[|]")).saveAsTextFile("validLogs ")
```

代码第 1 行是读入 logs 目录下的所有日志文件生成 RDD，并通过参数 partitionNum 指定生成的 RDD 的分区数。代码第 2、3 行读入规则，并生成一个广播变量发送到所有执行节点中。代码第 4 行先按照“|”符号分隔的方式提取日志每条记录中的字段，并通过检查字段数量是否为 67 判断每行记录的格式合法性，然后对第 50 个字段即用户访问的 URL 进行规则匹配。代码第 5 行将符合规则的记录输出到结果目录中。

我们使用 spark-shell 命令携带配置参数 num-executors 为 20 及参数 executor-cores 为 8 启动以上程序。即指定使用的系统资源为 20 个 Executor，每个 Executor 有 8 个 CPU 核，因此共计有 160 个 Executor Cores 供程序使用。输入数据为 1GB，规则文件的正则表达式为 21878 条，匹配后的输出文件为 1.3MB。当我们分别设置读入日志文件后创建 RDD 的分区数为 10、50、100 和 150 时，程序执行的时间如表 6-3 所示。

表 6-3 不同分区的程序性能测试

编号	分区数 partitionNum 参数	执行时间
1	10	2 小时 30 分
2	50	54 分
3	100	32 分
4	150	21 分

从实验结果我们可以看到，在可使用的计算资源为 160 个 Executor Cores 的情况下，当分区数过少时（例如 10 和 50），程序不能充分利用集群提供的计算资源，因此程序执行时间较长。当我们通过增加分区数量从而提高程序并行度时，则可以提升对计算资源的利用率，不断降低程序的执行时间，实现了更高的计算性能。

6.3 利用持久化

Spark 集群是一个通过网络连接各服务器节点的分布式处理系统,因此与其他分布式处理系统一样,网络通信开销也是决定 Spark 程序运行效率的重要因素之一。对于所有宽依赖(请参见 3.2.5 节)的 RDD 算子,由于子 RDD 的产生要依赖所有父 RDD,因此,这些算子的变换过程不能在一个计算节点中完成,而是要按照一定规则将多个节点中符合一定条件的一组 RDD 通过网络传送到集群中不确定的另一个节点进行计算,在此过程中还要进行 Shuffle。因此,对于使用了宽依赖算子的程序,如果规划不当,可能会因为过多的网络通信开销而影响程序性能。为了更清楚地展示这一问题,我们以一个具体的网络流量分析实例来说明。

假设我们可以获取一个网站的用户信息和访问网络日志。其中用户信息数据中包含了数以百万计的用户记录,其格式和内容如下所示。

```
userInfo.csv
```

```
86158… 2 3 …
86137… 2 1 …
86135…1 3…
```

在 userInfo.csv 中保存了每个用户的基本信息,其中第一列为用户 ID,第二列为用户所在的省份 ID,第三列为地市 ID,其他列还包含用户姓名、性别等更多个人信息,在这里我们暂时忽略。另外,我们还可以通过网络流实时获得用户通过 HTTP 协议访问网站的日志,该日志每 5 分钟产生一个,文件名为 accessLog_YYYYMMDDHHMM.csv,文件名“_”后的后缀是由文件产生时的年月日时分构成的,文件格式如下。

```
accessLog_*.csv
```

```
20151122120031 86158… qq 205 …
20151122120103 86137… qq 704 …
20151122120115 86135… baidu 235 …
…
20151122120401 86158… sp3 …
20151122120412 86135… sp1 …
20151122120456 86137… sp2 …
```

文件中保存了 5 分钟内每次访问的时间(为简单起见我们假定精确到秒)、用户

ID、被访问的网站（例如 qq、baidu 等）、流量（KB）以及其他我们暂时忽略的信息。我们分析的目标是要统计每 5 分钟不同省份对各个网站访问产生的流量。为此，我们编写了以下代码进行分析。

统计不同省份对各网站访问流量的示例程序（初始版本）

```
package io.github.lsrbupt.spark.performance
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD
object PrepartitionTuningRefer {
    def main(args: Array[String]): Unit = {
        if (args.length < 2) {
            println("Usage:<userInfoFile><accessLogPath><output>")
            System.exit(2)
        }
        val conf = new SparkConf().setAppName("Prepartition Tuning Refer")
        val sc = new SparkContext(conf)
        val userInfo = sc.textFile(args(0))
        val accessLogPath = args(1)
        val PHONE_NUM = 0
        val TOTAL = 10
        val unprepartitionUserInfo = userInfo.map(_.split("\t")).filter(_.length == TOTAL).map(fields => (fields(PHONE_NUM), fields)).persist
        val beginTime = "2015112212"
        for (i <- 0 until 12) {
            trafficCount(accessLogPath + "/accessLog_" + f"$beginTime${i * 5}%02d" + ".csv", sc, unprepartitionUserInfo, args(2))
        }
        sc.stop
    }
    def trafficCount(file: String, sc: SparkContext, userInfo: RDD[(String,Array[String])],output:String):Unit = {
        val PHONE_NUM = 1
        val TOTAL = 12
        val accessLog = sc.textFile(file, 2).map(_.split("\t")).filter(_.length == TOTAL).map(fields => (fields(PHONE_NUM), fields))
        val mergeInfo = accessLog.join(userInfo)
        val USER_PROVINCE = 1
        val SP = 2
        val UP_BYTES = 10
        val DOWN_BYTES = 11
        val trafficCount = mergeInfo.map({
            case (phoneNum, (accessLog, userInfo)) => (userInfo(USER_PROVINCE) + "\t" + accessLog(SP), accessLog(UP_BYTES).toLong + accessLog
```

```
(DOWN_BYTES).toLong)
35:         })
36:         .reduceByKey(_ + _)
37:         .map({case (k, v) => k + "\t" + v})
38:         trafficCount.saveAsTextFile(output + "/" + file.split("/").
last)
39:     }
40: }
```

代码 7 ~ 10 行读取命令行参数，参数 1 为用户信息数据文件路径，参数 2 为用户访问日志路径，参数 3 为结果输出路径。代码 11 ~ 14 行创建 SparkContext 并读取用户信息数据到 RDD 中，命名为 userInfo。代码 15 ~ 17 行先将 userInfo 中字段数目不正确的记录过滤掉，再把记录转换为键值对格式，键是用户手机号，值是记录，最后将输出的 RDD 持久化，使得之后不会重复进行这些变换。代码 18 ~ 21 行调用 trafficCount( ) 方法对 2015-11-22 12:00 开始的 1 小时用户访问记录（共 12 个文件，每 5 分钟一个）进行流量统计。代码 22 ~ 23 行停止 SparkContext 并退出程序。代码 24 ~ 39 行定义了方法 trafficCount( )。该方法接收四个参数，参数 1 是用户访问日志单个 5 分钟文件的路径，参数 2 是 SparkContext，参数 3 是经过清洗和格式转换之后的用户信息数据 RDD，参数 4 是结果输出路径。代码 25 ~ 27 行读取 5 分钟用户访问日志到 RDD 中，同时过滤掉字段数目不正确的记录，再把记录转换为键值对格式，键是用户手机号，值是记录。代码第 28 行将用户信息数据和用户访问日志以手机号为索引进行联结，将每一条用户访问网站记录填充上用户所在的省份信息。代码 29 ~ 36 行从联结后的每一条记录中提取出用户所在省份、访问的网站和总流量（上下行流量合并后的值），然后以前两者为键、流量为值进行归约。代码第 37、38 行将流量统计结果进行格式化并输出。代码的执行过程如图 6-3 所示。

初始版本程序在运行时，对于 accessLog 中每个 5 分钟文件，它都会提交一个 Job，该 Job 分为 4 个 stage，对应代码第 17、27、33 和 38 行。在实验中我们对每个 stage 进行了测量，第 1 个 Job 的各个 stage 运行时间分别为 11 秒、8 秒、24 秒和 1 秒，Shuffle 过程中经过 kryo 序列化之后的数据量为 165.6MB，其中 101.6MB 为 userInfo，64MB 为 accessLog。其中的 stage 1、2 和 3 都包含了 Shuffle 读写。我们注意到，对于每个五分钟 accessLog 文件的统计，要进行 join 的 userInfo 是不变的。如果将该数据集进行预分区并持久化，那么在处理完第一个 5 分钟文件后，该数据集就已经进行了一次 Shuffle 读写和数据传输。在接下来 11 个 Job 中，join 操作会知道 userInfo 已经完成 Shuffle 读写并持久化，不会再进行数据传输，因此可以减少 11 次 userInfo 的数据传输和处理开销。

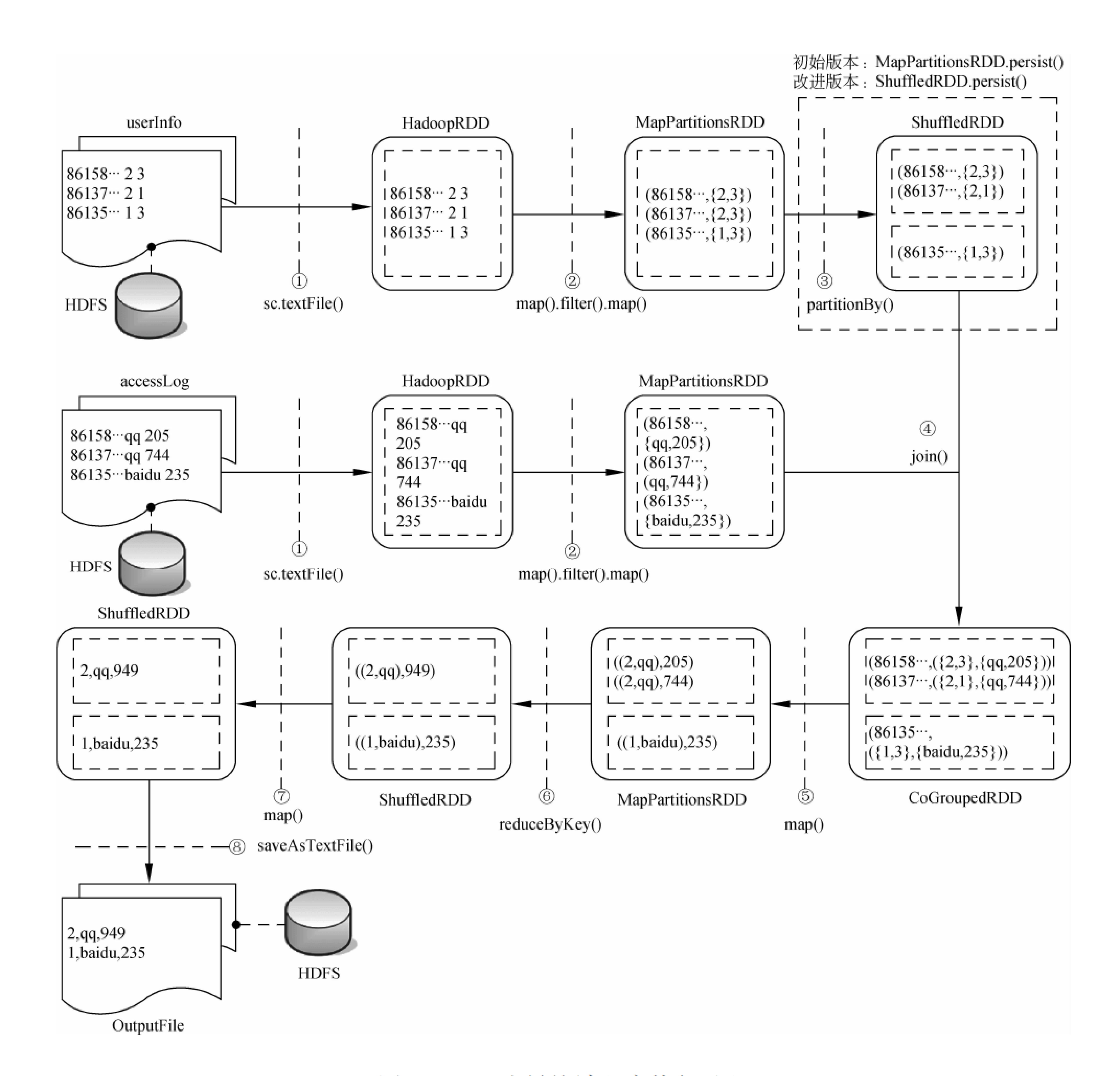


图 6-3 SP 流量统计程序执行过程

基于此分析，我们对代码的第 17、20 行进行如下改进。

统计不同省份对各网站访问流量的示例程序（改进版本）

```
17: val prepartitionUserInfo = userInfo.map(_.split("\t")).filter(_.length == TOTAL).map(fields =>
      (fields(PHONE_NUM), fields)).partitionBy(new HashPartitioner(24)).persist
…
20: trafficCount(accessLogPath + "/accessLog_" + f"$beginTime${i * 5}%02d" + ".csv", sc, prepartitionUserInfo,args(2))
```

我们在对用户信息数据进行清洗和格式转换后，将 userInfo 预分为 24 个分区并持久化到内存中。在进行流量统计时，trafficCount( ) 方法的参数 3 就是经过了清洗、格式

转换、预分区和持久化的用户信息数据 prepartitionUserInfo(初始版本则是未经过预分区的 unprepartitionUserInfo)。我们将初始版本和改进版本的两个程序在 2 个 Executor、每个 Executor 为 4 个 Cores 的配置下进行测试,得到的测试数据如表 6-4 所示。从测试数据我们可以看出,在对 userInfo 进行预分区并持久化后,参与 Shuffle 的数据量大幅减少(从 1842.5MB 减少到 647.1MB),并且程序执行时间明显降低(由 6 分 40 秒降低为 1 分 35 秒),程序性能得到了极大的提高。

表 6-4 预分区持久化性能对比

测试编号	是否预分区及持久化	Shuffle 数据量	执行时间
1	否	1842.5MB	6 分 40 秒
2	是	647.1MB	1 分 35 秒

6.4 选择恰当的算子

在本书的第 4 章,我们已经了解 Spark 提供了丰富的处理 RDD 的算子。我们在开发 Spark 程序时可以选择不同的算子组合来完成同样的功能,但显然并不是所有的实现方法都能获得一样的性能。因此我们在设计和实现 Spark 程序时需要注意不同算子的性能差异,挑选恰当的算子组合实现需要的功能,从而确保优异的 Spark 程序性能。从本书的 Spark 原理和 RDD 算子介绍中我们可以了解到,不同算子的差异主要影响的是 Shuffle 的次数和数据量,例如 repartition、join、cogroup、* By 和 * ByKey 等算子。由于 Shuffle 所涉及的数据都需要先写入磁盘,然后再通过网络传输完成计算,耗时较长,因此,我们需要重点关注这些算子的使用方式。为此,我们基于以往的开发经验整理了以下要点。

√ 避免使用 groupByKey 执行联合规约操作。例如,虽然 rdd. groupByKey( ). mapValues(_. sum)与 rdd. reduceByKey(_ + _)可以完成同样的加和计算功能,但是前一种实现方式需要在网络上传输整个数据集,而后一种方式先计算每个分区中每一个 Key 所对应 Value 的加和,然后在 Shuffle 后将这些加和合并为最后的结果。因此后一种实现方法比 groupBykey 的实现方法效率更高。我们还可以使用其他算子替代 groupByKey,例如,combinebykey 可以用来在返回类型不同于输入类型的结合键值对,foldbykey 可以预设一个规约函数和一个零值合并每个 Key 对应的 Value。

√ 当输入和输出的 value 类型不同时,避免使用 reduceByKey。例如,我们编写一

个转换用于查找对应于每个 Key 的唯一字符串，实现该转换的一种方法是使用 map 算子把每个键值对转换成一个集合，然后用 reduceByKey 结合，代码如下：

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2)).reduceByKey(_ ++_)
```

因为每个键值对需要分配一个新的集合，上述代码将创建大量不必要的集合对象。因此我们可以使用 aggregateByKey 来更加高效地在 map 端实现聚集，代码如下。

```
val zero = new collection.mutable.Set[String]()
rdd.aggregateByKey(zero)((set, v) => set += v, (set1, set2) => set1 ++=
set2)
```

√ 在进行关联时避免使用 flatMap-join-groupBy 算子的代码模式。当两个数据集已经按 Key 进行分组，可以使用 cogroup 来关联它们，这样可以避免解组和重新分组的开销。

√ 当不需要改变 Key 值时，避免使用 map 算子，而选用 mapValues 算子或 flatMapValues 算子。如果使用 map 算子对 Value 值对进行操作，传递给 map 算子的函数可能改变 Key，从而改变分区，导致 Shuffle。Spark 并不检查传递给 map 算子的函数是否改变 Value 值对应的 Key，而是提供 mapValues 和 flatMapValues 这两个不改变 Key 值的算子，以确保分区不发生改变。

为了直观展示不同算子对 Spark 程序的性能影响，我们用 groupByKey 和 reduceByKey 两个算子分别实现一个相同的服务器流量统计功能，并通过运行时间观察它们的性能差异。我们的测试数据是用户上网日志话单，其中字段 10 为一次数据流的开始时间(精确到毫秒的 UTC 时间)，字段 21 为服务器的点分十进制的 IP 地址，字段 24 为这次数据流的流量值。我们先使用 groupByKey 算子实现统计每台服务器每 10 分钟的流量和的功能，代码如下。

使用 groupByKey 算子统计服务器流量的示例代码

```
import java.text.SimpleDateFormat
import java.util.Date
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}
object SparkTest {
    def safeStringToDouble(str: String): Option[Double] = try {
        Some(str.toDouble)
    } catch {
        case e:NumberFormatException => None
    }
```

```
11:     def safeStringToLong(str: String): Option[Long] = try {
12:         Some(str.toLong)
13:     } catch {
14:         case e:NumberFormatException => None
15:     }
16:     def main(args: Array[String]): Unit = {
17:         val input = args(0)
18:         val output = args(1)
19:         val conf = new SparkConf().setAppName("SparkTest")
20:         val sc = new SparkContext(conf)
21:         val data = sc.textFile("/import_data/gd/20160115/103u1_154/").
map(x => x.split("\\|")).filter(_.length == 67).map(x =>(safeStringToLong(x
(9)),x(20),safeStringToLong(x(23)))).filter(x => x._1.nonEmpty&&x._3.
nonEmpty).mapPartitions(iter =>{
                    val df: SimpleDateFormat = new SimpleDateFormat("yyyy -
MM - dd HH:mm:ss")
                    iter.map(x => {
                        val utcTimeStamp = x._1.get.toLong /600000 * 600000
                        val timeString = df.format(new Date(utcTimeStamp))
                        (timeString + "," + x._2, x._3.get)
                    })}).groupByKey().map(t => (t._1,t._2.sum))
22:         data.saveAsTextFile("/output")
23:         sc.stop
24:     }
25: }
```

功能主要逻辑由上述代码的第 21 行实现。首先使用 textFile 读取输入文件，然后用 map 算子将输入文件映射成一行一行的数据。由于原来的日志文件是用“|”分割的，所以用 split 算子分隔并提取字段。在日志文件的定义中，全部字段数量应为 67 个。由于在数据传输中可能出现错漏，因此需要用 filter 过滤出格式正确的数据，然后将第 10 个字段（开始时间）、第 24 个字段（上行流量）转换成 Long 型。接着用 filter 算子剔除掉开始时间或上行流量为空的脏数据。由于第 10 个字段是精确到毫秒的 UTC 时间，而我们需要统计的是每 10 分钟内的上行流量和，所以将开始时间进行一个转换。通过将开始时间除以 600 000 再乘以 600 000 的方法获得以 10 分钟为单位的时间窗序号，序号相同的记录即为发生在同一个 10 分钟时间窗内的流记录。然后以时间窗序号和服务器 IP 组合为 Key，流量值为 Value，使用 groupByKey 算子把相同 Key 的数据分组集合，最后通过 map 算子将时间窗口和服务器 IP 相同键值对中的流量值相加，得到每台服务器每 10 分钟的上行流量和。代码的执行过程如图 6-4 所示。

同时，我们也设计了用 reduceByKey 算子实现同样功能的代码，其差异仅在第 21 行上，代码如下。

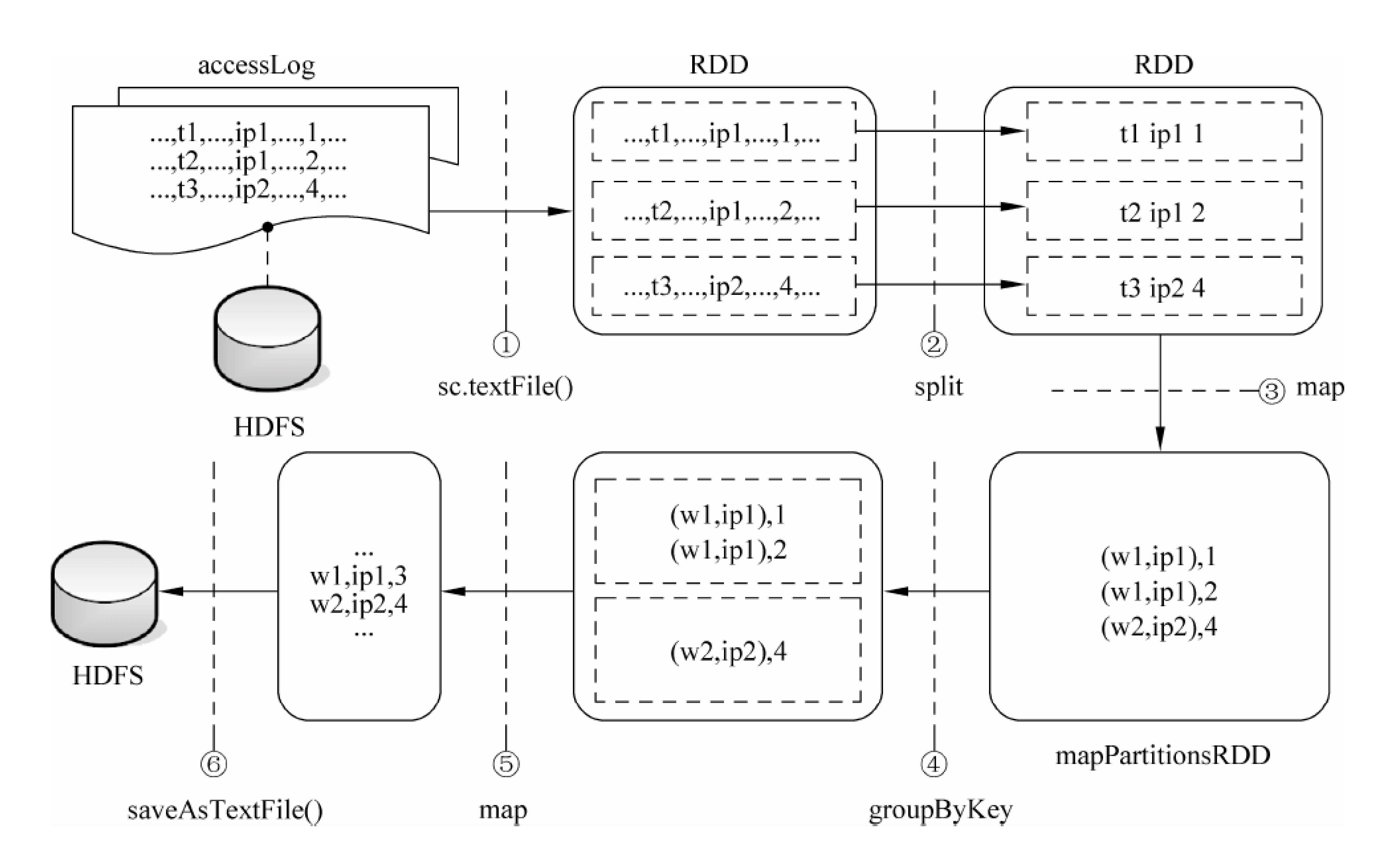


图 6-4 使用 groupByKey 统计服务器 10 分钟流量执行过程

使用 reduceByKey 算子统计服务器流量的示例代码

```
21:  val data = sc.textFile("/import_data/gd/20160115/103u1_154/").map
(x => x.split("\\|")).
        filter(_.length == 67).map(x => (safeStringToLong(x(9)), x(20),
safeStringToLong(x(23)))).
        filter(x => x._1.nonEmpty && x._3.nonEmpty).mapPartitions(iter => {
                val df: SimpleDateFormat = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss")
                iter.map(x => {
                val utcTimeStamp = x._1.get.toLong / 600000 * 600000
                val timeString = df.format(new Date(utcTimeStamp))
                (timeString + "," + x._2, x._3.get)
        })}).reduceByKey(_ + _)
```

使用 reduceByKey 统计服务器 10 分钟流量的执行过程如图 6-5 所示。

虽然这两个程序都能得到相同的统计结果，但是从执行过程图中我们可以清楚地看到两者的差异：

√ 使用 groupByKey 进行统计时，所有的键值对将被 Shuffle，这将导致很多不必要的数据在网络上传输。在 Shuffle 时，Spark 会调用一个分区函数确定每一个键值对应被 Shuffle 到哪一台机器上。当一台机器的内存不够时，Spark 还会将数据溢出到硬盘中，这将极大地影响程序的执行性能。

√ 使用 reduceByKey 时，在 Shuffle 之前，reduceByKey 可以让 Spark 知道对每个分

图 6-5 使用 reduceByKey 统计服务器 10 分钟流量执行过程

区上的每一个 Key 只需输出一个中间结果。通过传递给 reduceByKey 的自定义处理函数,在每个分区上具有相同 Key 的键值对在 Shuffle 前就被合并了。在 Shuffle 后,自定义处理函数再次被调用,将所有分区的中间结果进行规约,直接产生一个最终的结果。

可以想象,在处理一个很大的数据集时,reduceByKey 和 groupByKey 所需要 Shuffle 的数据量差别将变得很大。我们使用了一个大小为 405GB 的日志文件执行以上两个程序并通过 Web UI 观察 Shuffle 数据量和记录执行时间,测试数据如表 6-5 所示。从测试数据中我们可以明显地看出在完成相同功能时选择不同算子所造成的性能差异,使用 groupByKey 算子的 Shuffle 数据量是 reduceByKey 算子的 2 倍多,且使用 groupByKey 比使用 reduceByKey 算子多消耗了 14.3% 的时间。

表 6-5 reduceByKey 和 groupByKey 测试数据

测试编号	使用算子	Shuffle 数据量	执行时间
1	reduceByKey	4GB	7560 秒
2	groupByKey	9.3GB	8640 秒

6.5 利用共享变量

在本书第 3 章 Spark 原理部分我们已经了解到,一个 Spark 程序会被拆分为多个任务以分布式的方式运行在集群中的多个服务器上。在这一架构下,我们可以在任务内部定义及操作变量,这些变量只在任务内部有效,而不同任务之间的变量相互独立,

因此我们不能将任务中定义的变量作为 Spark 程序的全局变量使用。在实际应用中，我们有时候会需要能在任务间共享的全局变量。针对这种需求，Spark 提供了两种类型的共享变量：一种是用于汇聚信息的累加器，另一种是用于高效分发大型数据的广播变量。其中累加器为可读写变量，广播变量则为只读变量。下面我们对这两种类型共享变量分别进行介绍。

6.5.1　累加器变量

我们可以定义一个累加器变量，该变量只能在 Spark 程序的 Driver 进程上创建并取值。定义的累加器变量在作业的各个任务中完成计算后，会回传到作业的 Driver 进程中，然后对该变量继续累加得到最终结果。这一过程如图 6-6 所示。

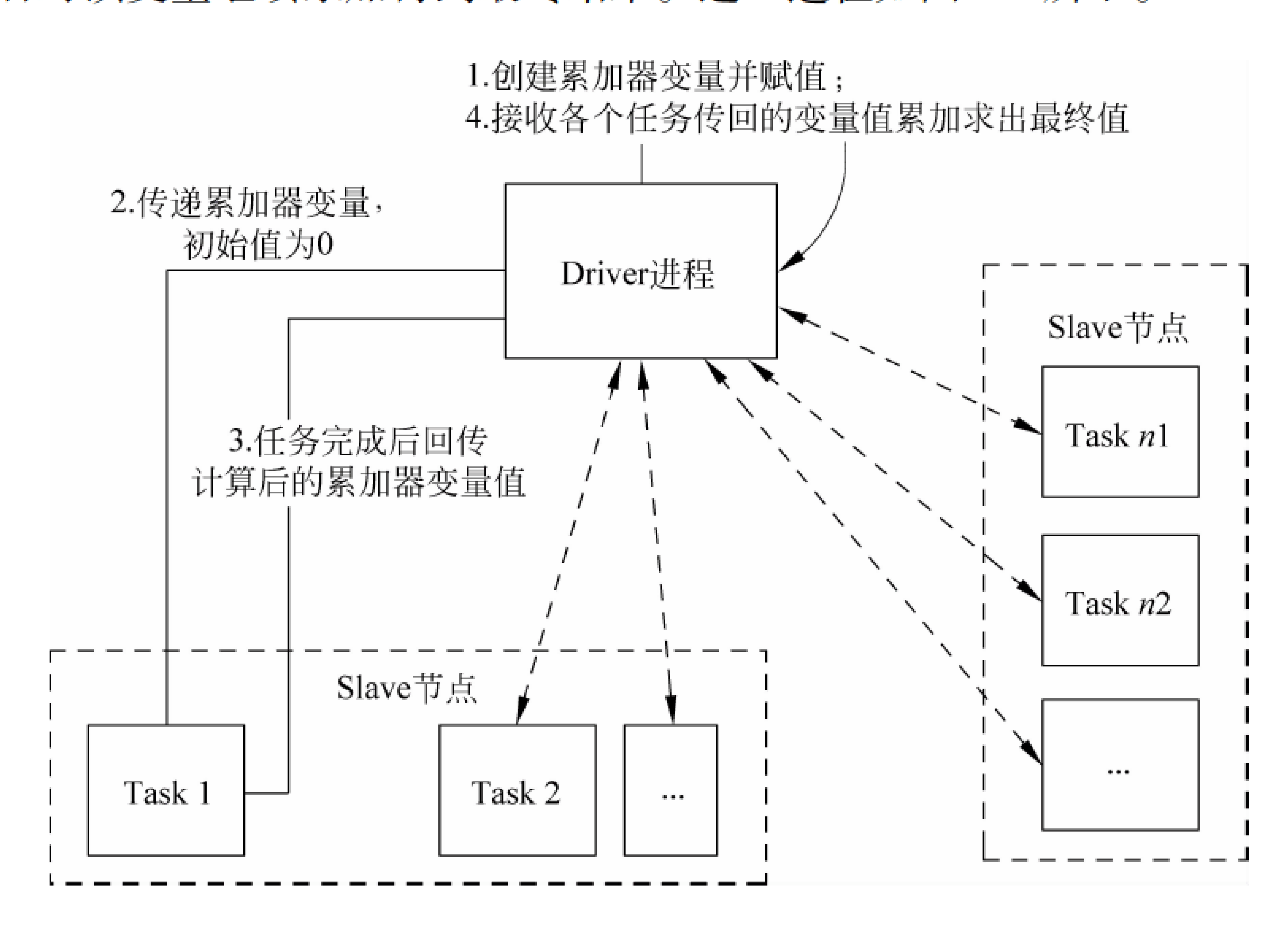


图 6-6　累加器变量的运行流程

（1）首先在 Driver 中定义累加器变量，并赋予一个初始值。

（2）Driver 将该累加器变量传递到各个执行任务的节点中。需要注意的是，传递过去的变量的初始值均为零，而不是我们在第（1）步中定义的初始值。

（3）当某个任务完成运行后，计算节点会将自己计算的累加器变量的值回传给 Driver 进程。

（4）Driver 将各个计算节点传回的值进行累加计算，得到最后的值。

需要注意的是，目前累加器只支持累加的操作，也可以用负数进行累加，但不支持减、乘、除等其他操作。在具体应用中，我们可以使用累加器对某个事件发生的次数进行追踪，特别是我们需要对程序中多种类型的事件同时进行跟踪时，累加器可以很好

地满足我们这样的需求。这里用一个简单的实例来进行说明。假设我们对采集到的日志文件使用 Spark 进行清洗时，想知道文件中有多少行的数据是无效数据，此时我们就可以通过累加器功能进行实现，其具体代码如下。

使用共享变量计算错误记录数的示例程序

```
1:   scala> val file = sc.textFile("data.csv")
     file: org.apache.spark.rdd.RDD[String] = data/ data1
MapPartitionsRDD[15]at textFile at <console>:21
2:   scala> val errorLines = sc.accumulator(0)
     errorLines: org.apache.spark.Accumulator[Int] = 0
3:   scala> val filterData = file.map(line => {
4:          | var lineData = line.split("\t")
5:          | var res = ""
6:          | if (lineData.length < 3) {
7:              | errorLines += 1
8:          | } else {
9:            | res = lineData(0) + "\t" + lineData(1) + "\t" + lineData(2)
10:           | }
11:         | res
12:   | })
     filterData: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[16]
at map at <console>:25
13:   scala> filterData.saveAsTextFile("filteredData")
14:   scala> println("error lines: " + errorLines.value)
      error lines: 20845
```

首先代码第 1 行从文件 data. csv 中读取数据，然后代码第 2 行使用 SparkContext. accumulator(0)方法来创建一个累加器变量 errorLines 并初始化变量值为 0。该语句执行后返回的结果是一个 org. apache. spark. Accumulator[Int]类型的对象，其中 Int 是初始值的类型。在后面对数据的处理过程中，如果输入的一行数据的列数少于3(代码第 6 行)，我们则认为是一行错误数据，此时对累加器变量 errorLines 加 1(代码第 7 行)。在代码第 14 行，我们在完成数据处理后打印出累加器的值，此时输出的是 Driver 程序将所有计算节点中返回的累加器变量 errorLines 相加后得到的最终值 20 845，即输入数据中一共有 20 845 条记录是错误数据。在 Spark 中 map 转换算子是惰性的，因此我们需要执行 saveAsTextFile 算子(代码第 13 行)后才能驱动该程序真正运行，此时累加器的操作才会随之执行并得到结果。当然，我们也可以使用其他行动算子来驱动程序运行。在实际应用中，累加器变量可以根据用户需要集成在不同规模不同粒度的 RDD 中进行计算。在输出累加器的值时，我们可以调用其 Value 属性。但需要注意的是，累加器的 Value 属性无法被 Worker 节点的任务获取。在 Worker 节点中，累加器变量仅

为只读变量。在这种机制下，累加器在 Worker 节点中只需要简单地执行累加操作即可，而不需要在 Worker 节点上更新导致额外的网络开销，从而确保高效运行。

目前 Spark 的累加器变量仅支持 double、float、int 和 long 类型，如果我们需要使用特殊的数据类型作为累加器，可以通过扩展 AccumulatorParam 类来增加自定义累加器类型，在此过程中需要使用 addInPlace、zero 和 addAccumulator 3 个方法，具体使用方法可以查询 Spark API 的官方文档。

6.5.2 广播变量

在进行数据分析时，我们经常会遇到这样一类场景，即部分关键数据可能在进行分布式处理时需要被每个处理节点都用到。例如，我们在做上网日志分析时，经常会以地域为维度分析服务器的请求来源。但是在日志文件中仅有访问用户的 IP 地址，并没有地域信息。要获得地域信息，需要通过一个单独的 IP 地址与地域信息对应的资源文件中的记录与上网日志记录的 IP 地址进行匹配来获得。为了完成这个匹配工作，一种简单的方法就是将资源文件中的数据封装成一个静态变量在程序中使用。这种方法虽然简单，但如果资源文件的数据量达到 GB 甚至更高级别时，封装静态变量的方法将变得非常低效。为了解决这一问题，Spark 提供了广播变量这一机制。广播变量用于将 Spark 程序中一个大型的只读变量发送到至每个 Slave 节点，供节点上的计算任务使用，其运行流程如图 6-7 所示。

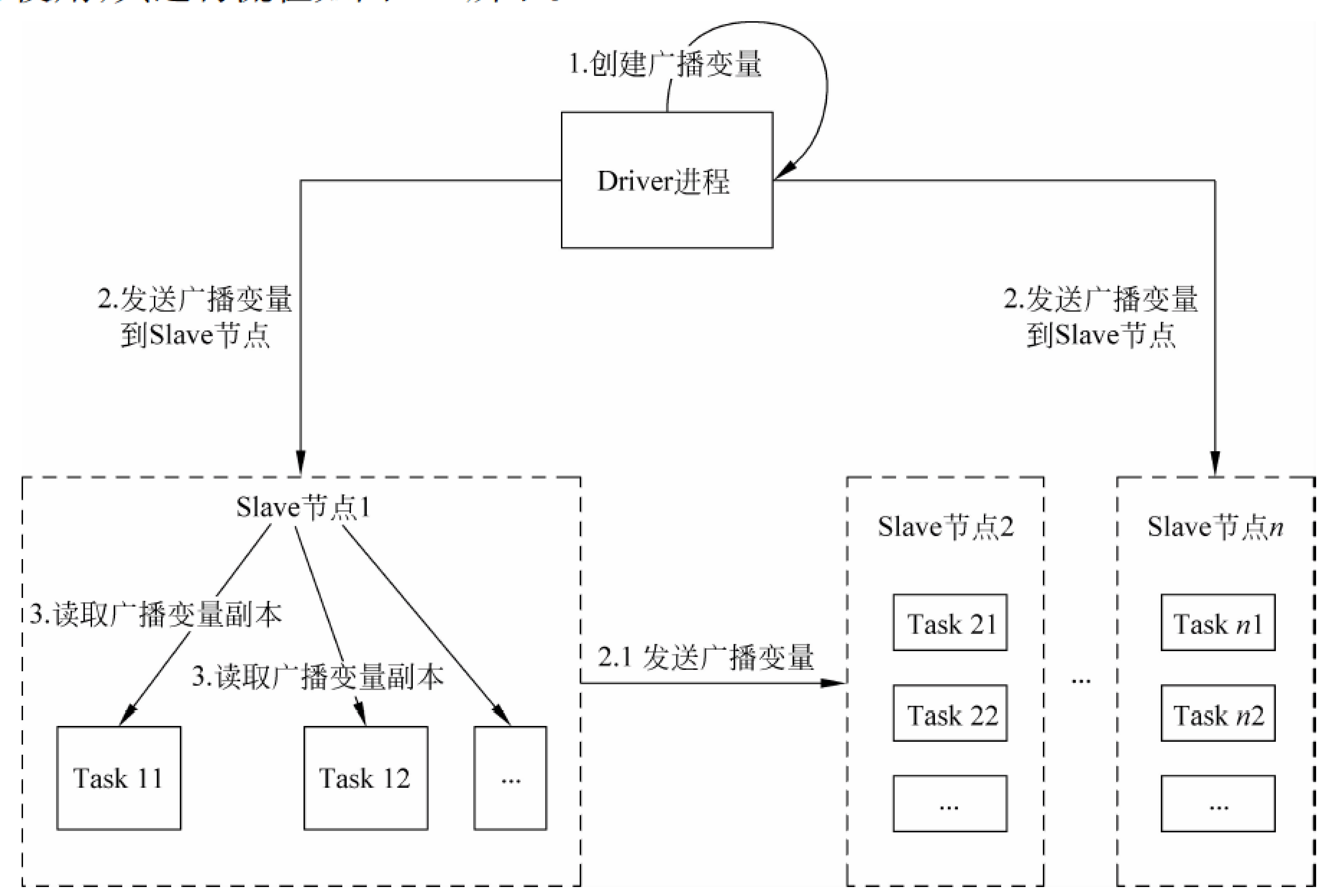


图 6-7 广播变量的运行流程

（1）首先在 Spark 程序中创建一个广播变量。

（2）Dirver 进程会将广播变量发送到各个 Slave 节点，当集群节点较多时，广播变量会采用 P2P 的方式来在节点之间分发广播变量副本（步骤 2.1）。

（3）当 Slave 节点的计算任务需要使用该广播变量时，会从该节点中读取该广播变量副本，而不需要通过网络从 Driver 进程中获取。

从广播变量的执行流程我们可以看到使用广播变量有多个好处：①广播变量只会发送一次到各个节点，然后节点上运行的任务需要使用广播变量时，均从该节点读取变量的副本进行操作，大大降低了数据传输量，提高任务运行速率；②广播变量为只读变量，保证了各个节点数据的一致性；③广播变量会采用 P2P 的通信机制，在集群节点较多的时候，该机制可以实现数据快速传播到各个节点，提高 Spark 程序的效率。

下面我们以按地域统计访问次数的功能为例展示广播变量的具体使用方法，代码如下所示（为简单起见，我们仅展示与广播变量相关的部分代码）。

使用广播变量按地域统计访问次数的示例程序

```
1: val ipLocationLib = sc.broadcast(loadIpLocationLib())
2: val countAccessByCity = accessLog.map{
3:        |case (ip, numbers) =>
4:        |val city = lookUpCity(ip, ipLocationLib.value)
5:        |(city, numbers)
6:   |}.reduceByKey(_ + _)
7: countAccessByCity.saveAsTextFile(outputFile)
```

代码的第 1 行将 IP 地址与城市的对应关系文件加载后的数据定义为广播变量 ipLocationLib。然后查询每行日志中的 IP 地址对应的城市（代码第 4 行）。并以城市为 Key 使用 reduceByKey 统计各个城市的访问次数（代码第 6 行）。

根据以往的经验，在使用广播变量时还有两点需要注意：①为节省空间，在广播变量使用完毕后，建议使用 unpersist 方法清理变量。但使用该方法时要特别注意，如果清除该值后还要再次调用，会引发该广播变量再次从 Driver 进程发送到各个 Slave 节点，因此只有在确定不再使用该广播变量后再进行清除；②如果某个变量包含的数据量太大，直接使用广播变量的简单数据格式会影响 Spark 程序运行效率，此时我们可以考虑先对该变量进行序列化再用广播变量的方式进行传播，序列化的具体使用方法可以参考下一小节。

6.6 利用序列化技术

我们在用 Spark 进行数据分析和数据挖掘时,往往伴随着大量的数据缓存和数据传输。在这个过程中,不论是 CPU、内存还是网络带宽,都有可能成为影响 Spark 程序性能的瓶颈。如果我们能够减少数据传输量和内存使用量,就可以为 Spark 程序的性能提升带来极大的好处。在 Spark 框架中,我们可以采用序列化的方式来存储 RDD,从而达到这一目的。

序列化是指将数据结构或者对象按照某种规则转化为可以快速存储或传输的形式(如二进制)的技术。与序列化相对应,反序列化则是将序列化后的数据转换为数据结构或者对象。图 6-8 是使用序列化与反序列化的典型场景。数据的提供方在对数据结构或者对象序列化后,将数据进行缓存或通过网络传输到数据的接收方。接收方从缓存中读取数据或者通过网络接收数据再对二进制字节流之类的数据进行反序列化操作,得到原始的数据结果或对象。序列化和反序列化是我们在程序开发中经常要使用的技术,在分布式、大数据应用中尤为重要,合理的序列化协议不仅可以提高应用的健壮性能和优化性能,还可以让应用更容易扩展和调试。

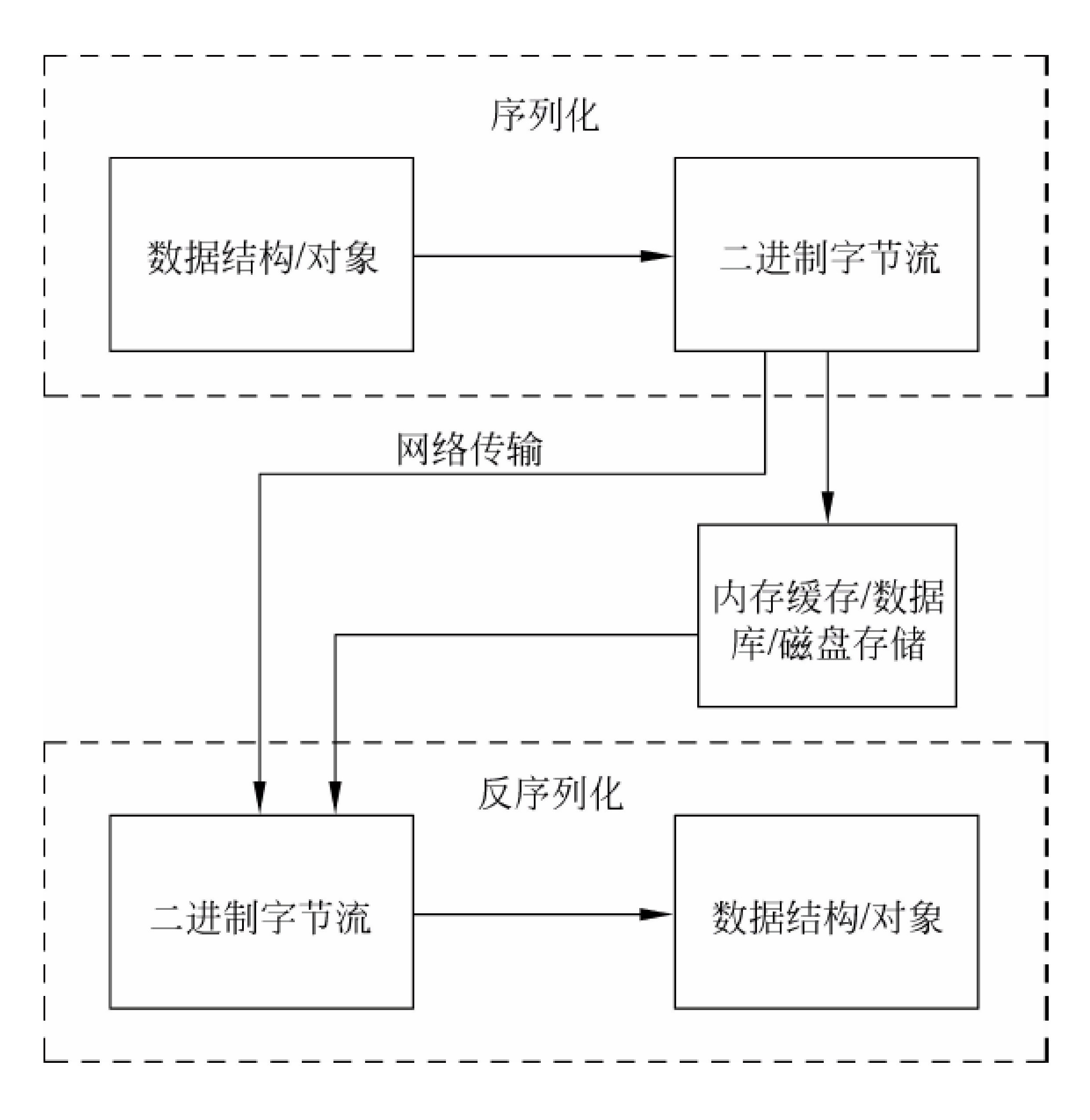


图 6-8 典型的序列化与反序列化应用场景

在利用序列化技术优化 Spark 程序时,我们主要从两方面考虑如何改善性能:时间和空间。时间开销主要体现在序列化和反序列化过程中的时间长短。空间开销主

要体现在序列化过程中增加的额外存储空间开销。当前的 Spark 框架中内置两种序列化方式,Java Serialization 和 Kryo Serialization。为了到兼容历史版本,当前 Spark 框架默认采用 Java 的 ObjectOutputStream 框架,它支持所有继承于 java. io. Serializable 的序列化方法。在实际 Spark 程序开发过程中,我们推荐使用 Kryo 进行序列化,因为该方式在压缩率和运行效率两方面都优于默认的 Java 序列化方式。不过需要特别注意的是,目前 Spark 任务的序列化还只支持 Java 序列化的方式,它通过 spark. closure. serializer 来配置,而其他数据的序列化如 Shuffle、RDD 缓存等均可使用 Kryo 序列化。

为使用 Kryo 序列化,我们首先需要在应用的 SparkConf 配置中进行注册,例如 conf. set(" spark. serializer" , " org. apache. spark. serializer. KryoSerializer")。然后我们可以使用 registerKryoClasses 方法来注册自定义 Kryo 类,在后续 Shuffle 或者 RDD 缓存中使用 Kryo 进行序列化。下面我们以一个具体实例来说明如何使用 Kryo 序列化方式,并和 Java 序列化的性能进行对比。该实例的功能是对网络访问日志进行分析,统计每个用户在 5 分钟内的上下行流量,代码如下。

统计中每个用户每 5 分钟内的上下行流量的实例程序

```
1:   val conf = new SparkConf().setAppName("TrafficCount")
2:    conf. set ( " spark. serializer ",  " org. apache. spark. serializer.
KryoSerializer")
3:   val sc = new SparkContext(conf)
4:   val file = sc.textFile(args(0))
5:   val recordToArray = file.map(_.split("\t"))
6:   val filterRecords = recordToArray.filter(_.length == 87)
7:   val mapOutput = filterRecords.map(fields => {
8:       val minute = fields(BEGIN_TIME).toDouble.toInt / 300 * 300
9:       val key = fields(USER_ID) + "\t" + minute
10:      (key, (fields(UPBYTES).toLong, fields(DOWNBYTES).toLong))
11:  })
12:  val reduceOutput = mapOutput.reduceByKey((x, y) => (x._1 + y._1, x._2
 + y._2), args(2).toInt)
13:  reduceOutput.saveAsTextFile(outputFile)
```

代码第 2 行通过 SparkConf 的 set 接口配置使用 Kryo 序列化方式,如果不包含这行语句,则使用默认的 Java 序列化方式。代码第 4、5 行从程序参数指定的文件中读取数据,并用制表符进行字段分隔。代码第 6 行通过每行数据的字段数量是否等于 87 进行数据合法性判断。代码第 8 行以精确到秒 UTC 时间为格式的流起始时间(由常量 BEGIN_TIME 指定字段序号)转换为 5 分钟窗口序号,例如 UTC 秒值 1453113613(时间为 2016/ 1/ 18 18:40:13)会被转换为 1453113600(时间为 2016/ 1/ 18 18:40:

00)。代码第 9、10 行生成 Key/ Value 对,Key 值为用户 ID 加时间窗序号,Value 值为上行流量和下行流量构成的 Tuple。代码第 12 行基于 Key 值进行 reduce 操作,将相同 Key 值的多个 Value 值相加,即对上行流量和下行流量分别求和。代码第 13 行将结果输入到文件中。

我们使用上述程序,分别测试使用 Kryo 和 Java 两种序列化方式的性能。输入文件大小均为 296.5GB,运行结果也一致。使用资源为 20 个 Executor,每个 Executor 的 CPU 核数为 3。为避免集群的背景负载波动导致测试结果偏差,对每种序列化方式的程序我们分别运行了两次,取平均时间。测试结果如表 6-6 所示。我们可以看到采用 Kryo 序列化方式后,Shuffle 数据量有明显的缩小,且运行效率有一定的提升。

表 6-6　Java 与 Kryo 序列化性能比较

测试序号	序列化方式	Shuffle 数据量	运行时间
1	Java	7.4GB	198 秒
2	Java	7.4GB	192 秒
3	Kryo	4.7GB	186 秒
4	Kryo	4.7GB	174 秒

6.7 关注数据本地性

我们知道,在处理大量数据时,需要使用由多台服务器构成的集群运行 Spark 程序。在集群环境下,一个计算节点所要使用的数据有可能来自于集群中的任何一台服务器,例如本机、机架中的另一台服务器或其他机架的一台服务器。当输入数据和计算任务在同一个节点,通过本地磁盘 IO 即可得到数据。而当数据与任务处在不同的服务器时,则需要耗费一定的时间来通过网络传输数据。因此,在不考虑其他因素的情况下,从以上 3 种情况获取数据所需的时间显然是依次增加的。由此可以看出,节点获取计算所需数据的位置这一因素,对 Spark 程序运行的性能有着很大的影响,这一因素就称为数据本地性(Data Locality)。Spark 中定义了 5 种数据本地性级别,按照数据与任务距离的远近定义如下。

(1) PROCESS_LOCAL:数据和代码在同一个 JVM 中,这是最优的情况。

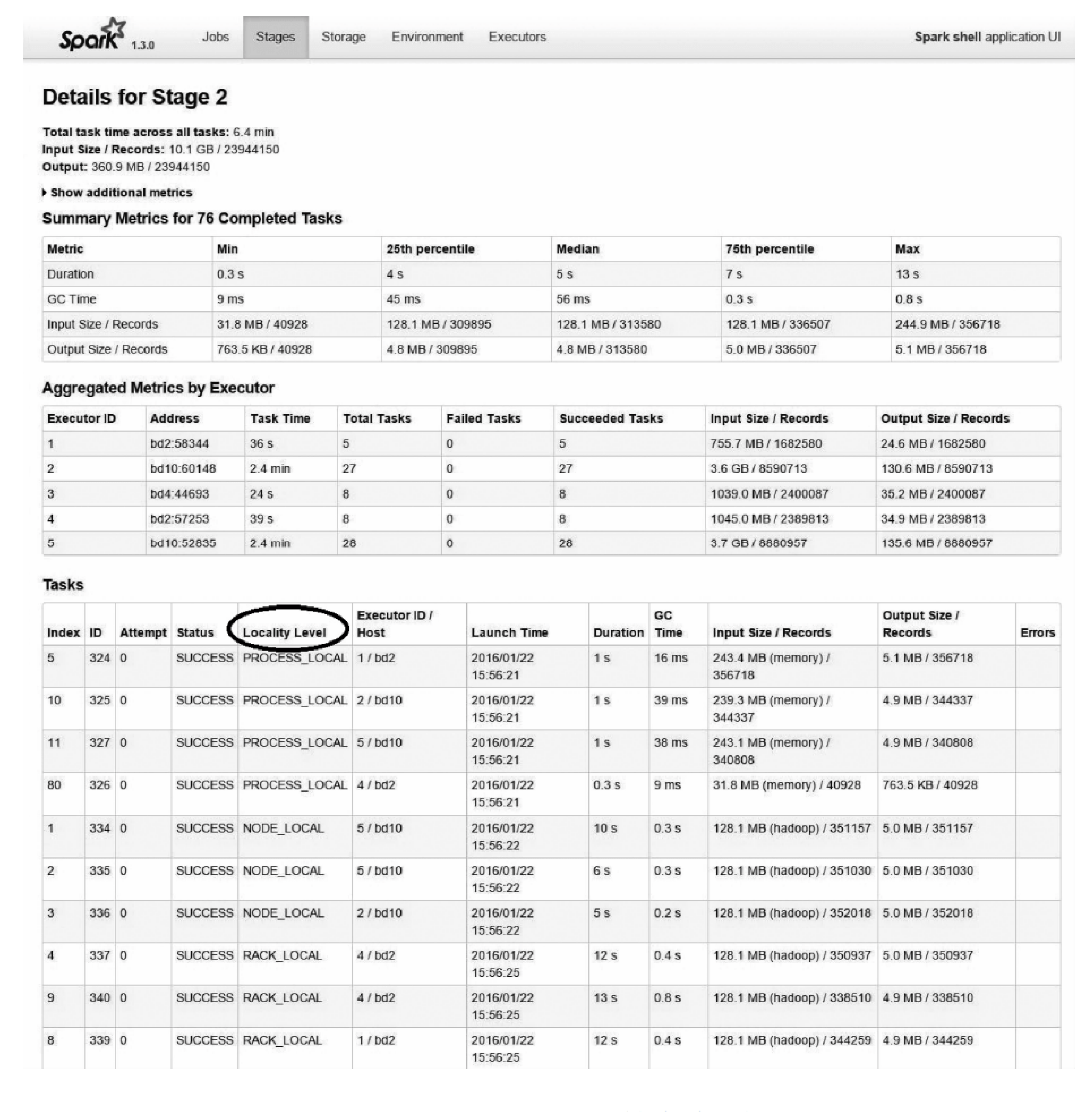
(2) NODE_LOCAL:数据和代码在同一台机器上,例如在同一台机器的 HDFS 中,或者是同一台机器的其他 Executor 中。该情况要比 PROCESS_LOCAL 稍微慢一些。

(3) NO_PREF:指数据在任何节点上获取都是一样的速度,例如将一个 Driver 中的 collection 对象变成 RDD。

（4）RACK_LOCAL：这种级别是指数据在同一个机架的其他一台机器上，获取这些数据一般需要通过一个交换机。

（5）ANY：这个级别是指数据要跨机架传输。

我们在运行 Spark 程序时，可以从 Spark 的 Web UI 中查看该程序运行时的数据本地性情况。图 6-9 展示了一个实例，通过图中的“Locality Level”这一列可以看到各个任务的数据本地性级别。我们可以看到，当数据本地性级别为 PROCESS_LOCAL 时，任务运行速度普遍较快，运行时间明显少于数据本地性级别为 NODE_LOCAL 和 RACK_LOCAL 的任务。



Details for Stage 2

Total task time across all tasks: 6.4 min
Input Size / Records: 10.1 GB / 23944150
Output: 360.9 MB / 23944150

▸ Show additional metrics

Summary Metrics for 76 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.3 s	4 s	5 s	7 s	13 s
GC Time	9 ms	45 ms	56 ms	0.3 s	0.8 s
Input Size / Records	31.8 MB / 40928	128.1 MB / 309895	128.1 MB / 313580	128.1 MB / 336507	244.9 MB / 356718
Output Size / Records	763.5 KB / 40928	4.8 MB / 309895	4.8 MB / 313580	5.0 MB / 336507	5.1 MB / 356718

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Output Size / Records
1	bd2:58344	36 s	5	0	5	755.7 MB / 1682580	24.6 MB / 1682580
2	bd10:60148	2.4 min	27	0	27	3.6 GB / 8590713	130.6 MB / 8590713
3	bd4:44693	24 s	8	0	8	1039.0 MB / 2400087	35.2 MB / 2400087
4	bd2:57253	39 s	8	0	8	1045.0 MB / 2389813	34.9 MB / 2389813
5	bd10:52835	2.4 min	28	0	28	3.7 GB / 8880957	135.6 MB / 8880957

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Output Size / Records	Errors
5	324	0	SUCCESS	PROCESS_LOCAL	1 / bd2	2016/01/22 15:56:21	1 s	16 ms	243.4 MB (memory) / 356718	5.1 MB / 356718	
10	325	0	SUCCESS	PROCESS_LOCAL	2 / bd10	2016/01/22 15:56:21	1 s	39 ms	239.3 MB (memory) / 344337	4.9 MB / 344337	
11	327	0	SUCCESS	PROCESS_LOCAL	5 / bd10	2016/01/22 15:56:21	1 s	38 ms	243.1 MB (memory) / 340808	4.9 MB / 340808	
80	326	0	SUCCESS	PROCESS_LOCAL	4 / bd2	2016/01/22 15:56:21	0.3 s	9 ms	31.8 MB (memory) / 40928	763.5 KB / 40928	
1	334	0	SUCCESS	NODE_LOCAL	5 / bd10	2016/01/22 15:56:22	10 s	0.3 s	128.1 MB (hadoop) / 351157	5.0 MB / 351157	
2	335	0	SUCCESS	NODE_LOCAL	5 / bd10	2016/01/22 15:56:22	6 s	0.3 s	128.1 MB (hadoop) / 351030	5.0 MB / 351030	
3	336	0	SUCCESS	NODE_LOCAL	2 / bd10	2016/01/22 15:56:22	5 s	0.2 s	128.1 MB (hadoop) / 352018	5.0 MB / 352018	
4	337	0	SUCCESS	RACK_LOCAL	4 / bd2	2016/01/22 15:56:25	12 s	0.4 s	128.1 MB (hadoop) / 350937	5.0 MB / 350937	
9	340	0	SUCCESS	RACK_LOCAL	4 / bd2	2016/01/22 15:56:25	13 s	0.8 s	128.1 MB (hadoop) / 338510	4.9 MB / 338510	
8	339	0	SUCCESS	RACK_LOCAL	1 / bd2	2016/01/22 15:56:25	12 s	0.4 s	128.1 MB (hadoop) / 344259	4.9 MB / 344259	

图 6-9　通过 Web UI 查看数据本地性

数据本地性级别对于 Spark 作业运行速度有很大的影响。如果数据和代码在同一台机器上面，那么就可以很快地运行。但是如果不在同一台机器上，那么就需要移动其中一个。一般来讲，由于代码的数据量较小，移动代码要比移动数据更快。因此

Spark 一般倾向于尽量少移动数据,让所有的任务都运行在最好的数据本地性级别环境中。然而,并不是每个任务都能达到这种要求。当在一个 Executor 上没有需要处理的数据而空闲之后,Spark 就会处理更低本地性级别的数据。此时,Spark 有以下两种策略来启动任务:①不移动数据,等待之前的任务结束,然后再启动任务;②移动数据以在空闲的 Executor 上运行任务。第一种方式减少了数据的移动,但是可能造成 CPU 资源的浪费。第二种方式则是可以充分利用 CPU 资源,但是可能会因为移动数据太多导致过多的 IO 操作。那么 Spark 是如何在移动数据和充分利用 CPU 之间进行权衡的呢?Spark 的策略是先等待一小段时间,等待的目的还是希望有 CPU 空闲出来以减少数据移动。如果有 CPU 空闲出来,那么就直接启动任务,不需要移动数据。如果没有,那么就移动数据到其他空闲的 Executor 上面运行新的任务。这种方式称为延时调度算法。该算法在切换数据本地性级别的时候,通过付出等待一小段时间的代价,实现利用数据本地性和利用空闲 CPU 资源的均衡,达到提高集群使用率的目的。切换数据本地性级别时等待的时间可以通过 spark. locality. wait 参数来进行统一的配置,也可以通过 spark. locality. wait. process、spark. locality. wait. node、spark. locality. wait. rack 等参数分别配置,具体配置选项如表6-7 所示。需要注意的是,如果我们配置的等待时间过长,也有可能导致我们提交的 Spark 程序运行时间加长,降低程序的运行效率。因此,在配置时需要根据我们的集群环境和作业类型来具体评估,以提高数据本地性任务的比率,有效地减少网络传输的数据量,降低磁盘 IO 的消耗,从而提高 Spark 程序的执行效率。

表 6-7 数据本地性相关配置选项

属 性	含 义	默 认 值
spark. locality. wait	当前数据本地性级别任务的等待时间,超过该时间后 spark 会选择低一个级别的任务运行。当用户的 Spark 程序任务运行时间较长同时本地性任务较少时,需要增大该参数数据值	3 秒
spark. locality. wait. node	启动数据在同一节点的任务的等待时间,超时后 spark 会选择数据在同一机架的任务运行	与 spark. locality. wait 相同
spark. locality. wait. process	启动数据在同一 Executor 进程的任务的等待时间,超时后 spark 会选择数据在同一节点的任务运行。该属性会影响在一个特定 executor 进程中尝试访问缓存数据的任务执行	与 spark. locality. wait 相同
spark. locality. wait. rack	启动数据在同一机架的任务的等待时间,超时后 spark 会选择数据在其他机架的任务运行	与 spark. locality. wait 相同

另外,除了通过配置上述数据本地性等待时间的参数外,我们还可以从其他方面来提高数据本地性任务的比率。例如：当数据副本太低时适当增加数据的副本数；通过适当地使用 cache 缓存数据来提高数据本地性级别为 PROCESS_LOCAL 的任务数量；对程序本身进行优化等。

为了直观地展示数据本地性对 Spark 程序性能的影响,我们利用文本计数程序设计了一个简单的实验。该试验的输入数据为一个大小 30GB 的文本文件 data. csv,我们使用一行简单的代码进行计数：sc. textFile("/ log/ data. csv"). count。为了实现不同的数据本地性效果,我们配置程序运行的总核数保持为 9、总内存保持为 9GB,但 Executor 数量分别为 1、3、9,测试结果如表 6-8 所示。

表 6-8　数据本地性性能测试数据

Executors	Executor Cores	Executor Mem	NODE_ LOCAL	RACK_ LOCAL	ANY	执行时间
1	9	9GB	5	72	44	276 秒
3	3	3GB	14	104	3	128 秒
9	1	1GB	64	57	0	56 秒

从测试结果我们可以看出,数据加载后生成的 RDD 一共有 121 个分区。当只有 1 个 Executor 时,仅有 5 个分区能从本机读取,其他的分区均需要从其他服务器上读取,其中 72 个分区从同一机架的服务器中读取,44 个分区需要跨机架读取。当 Executor 数为 3 时,更多的(14 个)分区可以从本机读取,从同一机架读取的分区数也增加到 104,只有 3 个分区需要跨机架读取。最后当 Executor 数增加到 9 时,可以看到有 64 个分区从本机读取,57 个分区从同一机架的服务器读取,并且没有跨机架读取的分区。由于本机读取、同一机架读取、跨机架读取数据的耗时是依次递增的,这也是随着 Executor 数量的增加执行时间逐渐减少的原因。

6.8　内存优化策略

Spark 中的内存使用分为两类：执行内存和存储内存。执行内存指的是在执行 Shuffle、连接、排序和聚集等操作时用于计算的内存,而存储内存指的是用于 Cache 和广播到集群中的数据所使用的内存。为了管理这两类内存,Spark 框架中定义了以下原则：①Spark 中执行内存和存储内存共享一个内存空间(我们定义为 M)；②当没有存储内存被使用时,Shuffle 等操作功能可以申请使用所有的可用内存；③当没有执行内存被使用时,存储功能(Cache 和广播)可以申请使用所有的可用内存；④在执行

Shuffle 等算子操作时，可能会在必要时将存储内存清除以供算子操作使用，但在总存储内存中的一个指定比例的内存空间（定义为 R）不受影响，即 R 定义了 M 中一个固定大小的子区域，存储在该区域中的数据块不会被算子操作功能所清除；⑤与此相反，在执行 Cache 和广播时，不能清除执行内存中的数据。

Spark 框架中的这些原则确保了内存管理的几个优良特性：①不使用 Cache 和广播的 Spark 程序可以将所有的内存作为执行内存，减少将数据从内存溢出到磁盘的低效操作；②使用 Cache 或广播的 Spark 程序可以保留一个较小的存储空间 R，在这个空间中的数据块不会被清除；③为 Spark 程序开发者提供了合理的开箱即用特性，开发者不需要关心内存在内部如何划分和管理等细节操作。

在 Spark 的这一内存管理框架下，我们在对 Spark 程序的内存使用进行优化时，首先需要确定数据集需要占用的存储空间。为了确定这一存储空间的大小，我们可以创建一个 RDD，并使用 cache 算子将它缓存在内存中，然后通过 WEB UI 中的"Storage"标签页查看该 RDD 占用的存储空间。图 6-10 展示了一个 RDD 使用的存储空间实例。其中"Storage Level"一列中的 3 个标签"Memory""Deserialized""1x Replicated"，分别表示被缓存的 RDD 只缓存在内存中、不进行序列化、没有额外的副本。"Cached Partitions"一列表示 RDD 有 11 个分区被缓存了。"Fraction Cached"一列表示该 RDD 被缓存的比例。后面 3 列分别表示该 RDD 存储在内存、Tachyon 内存文件系统和硬盘上的大小，我们可以看到该 RDD 被 100% 缓存了，同时由于数据会被多份缓存，所以 1.4GB 的RDD 占用了 2.8GB 内存空间。而在 Tachyon 和硬盘上的占用的空间大小均为 0B。

Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
Memory Deserialized 1x Replicated	11	100%	2.8 GB	0.0 B	0.0 B

图 6-10 Spark Shell 的 UI 中 storage 页面

在完成内存使用量预估后，我们可以从多个方面着手进行内存优化。

√ 通过数据结构的设计减少数据对象大小。

在设计数据对象的数据结构时，我们要尽量避免使用 Java 中会额外增加开销的数据结构，包括：

（1）优先使用数组和基本数据类型，而不是标准 Java 或者 Scala 的容器类（如 HashMap）。例如我们可以利用一个与 Java 标准库兼容的支持基本数据类型的容器库 fastutil（http://fastutil.di.unimi.it/）。

（2）尽可能避免使用基于引用的数据结构和封装类型的对象，以及包含大量小对象和引用的嵌套结构。例如，用 Int 数组，而不是 LinkedList。

（3）考虑使用数字的 ID 或者枚举类型对象来表示 Key，而不是使用字符串。

（4）如果内存小于 32GB，可以设置 JVM 参数“-XX：+UseCompressedOops”，使得引用使用 4 个字节而不是 8 个字节。该参数可以在 conf/ spark-env. sh 中设置，通过添加“export SPARK_JAVA_OPTS = '-XX：+UseCompressedOops”命令实现。

√ 通过序列化减少数据传输。

在优化了对象的数据结构后，如果 Spark 程序中的对象仍然很大，就需要通过序列化的方法减少数据传输量。例如，在对 RDD 进行持久化或缓存时，选择包含序列化选项的存储级别，如 MEMORY_ONLY_SER（具体参见第 4 章的 persist 算子描述），同时建议选择效率更高的序列化库（例如本章第 6 节介绍的 Kryo）。

√ 优化 GC。

众所周知，JVM 的垃圾回收（Garbage Collection，GC）是影响程序性能的重要因素。由于 Spark 程序最终是转换为 Java 程序运行的，因此如果 Spark 程序中的 RDD，尤其是会被多次使用的 RDD 数据量较大时，GC 将成为影响 Spark 程序性能的重要因素。为了优化 GC，我们首先需要收集统计信息，以确定 GC 出现的次数是否频繁以及 GC 的时间开销如何。在启动程序的 spark-submit 命令后面添参数“--conf spark. executor. extraJavaOptions = '-verbose：gc -XX：+PrintGCDetails -XX：+PrintGCTimeStamps'”，这样 Executor 就会输出 GC 的信息到 Worker 节点的 stdout 日志中，以便我们进行有针对性的调试和优化。在完成 GC 统计信息的收集工作后，我们可以借鉴优化 Java 程序 GC 的方法，例如：①当 Spark 程序执行算子操作所使用的内存和缓存的 RDD 内存之间出现冲突时，可以通过设置参数 spark. storage. memoryFraction 来控制分配给 RDD 缓存的空间大小，以减轻这个问题。毕竟少缓存一些对象比过多 GC 而拖慢整个作业执行的速度要更好一些。②尽量使存活时间长的 RDD 存储在 Old 区域，并留出足够的空间在 Young 区域以存储生命周期较短的对象，这样可以避免 Full GC 去回收那些任务执行期间创建的临时对象，以提高性能。

为了更加直观地展示内存对 Spark 程序性能的影响，我们设计了一个简单的实例。我们生成了两个较大的文件，其中分别包含 5 亿和 7 亿个随机产生的整数，文件大小为 3.7GB 和 5.2GB，文件名为 5y. csv 和 7y. csv。我们使用以下代码测试 GC 对 Spark 程序性能的影响。

GC 对 Spark 性能的影响示例程序

```
1: scala> val rdd1 = sc.textFile("/user/qinchao/input_test/5y.txt")
   rdd1: org.apache.spark.rdd.RDD[String] = /user/qinchao/input_test/
5y.txt MapPartitionsRDD[1]at textFile at <console>:21
2: rdd1.count
   res0: Long = 500000000
3: scala> val rdd2 = sc.textFile("/user/qinchao/input_test/5y.txt").
cache()
   rdd2: org.apache.spark.rdd.RDD[String] = /user/qinchao/input_test/
5y.txt MapPartitionsRDD[3]at textFile at <console>:21
4: scala> rdd2.count
   res1: Long = 500000000
5: scala> val rdd3 = sc.textFile("/user/qinchao/input_test/7y.txt")
   rdd1: org.apache.spark.rdd.RDD[String] = /user/qinchao/input_test/
7y.txt MapPartitionsRDD[5]at textFile at <console>:21
6: scala> rdd3.count
   res2: Long = 700000000
7: scala> val rdd4 = sc.textFile("/user/qinchao/input_test/7y.txt").
cache()
   rdd1: org.apache.spark.rdd.RDD[String] = /user/qinchao/input_test/
7y.txt MapPartitionsRDD[7]at textFile at <console>:21
8: scala> rdd4.count
   res3: Long = 700000000
```

以上代码每两行为一次测试，分别对 5y. csv 和 7y. csv 进行无缓存和有缓存的计数操作。在测试中，我们将每个 Executor 分配的内存设置为 1.3GB，小于要处理的文件大小。测试结果如表 6-9 所示。我们可以看到，在相同处理数据量的情况下使用了 Cache 的执行时间均大于不使用 Cache 的情况。其原因在于，文件在 Cache 后占用的内存空间是原始大小的 2～3 倍，在内存不足的情况下，使用 Cache 会导致过多的 GC 影响程序执行性能。

表 6-9 GC 对 Spark 程序性能的影响

编号	文件	是否 Cache	执行时间 1	执行时间 2	执行时间 3	平均时间
1	5y. csv	否	84 秒	59 秒	54 秒	66 秒
2	5y. csv	是	204 秒	162 秒	156 秒	174 秒
3	7y. csv	否	132 秒	144 秒	132 秒	136 秒
4	7y. csv	是	300 秒	216 秒	180 秒	232 秒

6.9 集成外部工具

我们除了可以使用 Scala、Java 和 Python 来写 Spark 程序之外，还可以使用其他语言来编写数据分析程序。Spark 提供一种统一的机制来将数据通过管道的方式传递到用户使用其他语言开发的程序或者脚本中，如 Perl、R 语言脚本、bash 脚本等，这种机制称为 pipe 管道。通过 pipe 命令可以将 RDD 数据的每个分区传递到我们编写的脚本程序中，作为脚本程序的输入数据，同时将结果作为字符串标准输出。如果我们已经开发了一些用其他语言或工具编写的数据处理程序或算法，并且想与 Spark 程序结合使用，我们就可以利用 pipe 管道来进行整合。除管道外，Spark 在 1.4 版本之后还提供了另一强大组件——SparkR。SparkR 提供了一种轻量级的方式，使得可以在 R 语言中使用 Spark，以充分利用 R 语言中丰富的库函数和算法。SparkR 极大扩展了使用 Spark 完成和数据挖掘工作的能力，同时也可以解决 R 语言框架无法扩展适应大数据环境的问题。

SparkR 实现了分布式的 DataFrame，支持在 DataFrame 进行查询、过滤以及聚合等操作。DataFrame 是 SparkR 的核心数据模型。DataFrame 与 RDD 类似，是一个分布式数据容器。但 DataFrame 与 RDD 又有所区别，在 DataFrame 中除了数据之外还包含了数据的结构信息。我们用一个简单的网络流量日志数据结构直观展示 RDD 与 DataFrame 的区别，如图 6-11 所示。图 6-11(a) RDD 中只是由一条条网络流量日志构成的集合，这些记录的具体字段含义及结构由使用者自己在 Spark 程序中定义和解析。图 6-11(b) 为一个 DataFrame 的数据块，它不仅包含了流量日志的具体数据，还包含了流量日志的格式定义，包括用户 ID、时间戳、上行流量、下行流量大小。

RDD中记录形式

TrafficLog
TrafficLog
TrafficLog

(a)

DataFrame数据记录形式

userID	timestamp	trafficUp	trafficDown
String	Double	Double	Double
String	Double	Double	Double
String	Double	Double	Double

(b)

图 6-11 DataFrame 与 RDD 中数据记录的区别

DataFrame 不仅支持简单数据结构，也支持复杂的嵌套数据类型，例如 Struct、Array 和 Map。同时，DataFrame 对代码生成和内存管理等方面都进行了优化，使得 SparkR 的

性能与我们在 Spark 中使用 Scala、Java 或 Python 直接编写程序的性能相当。当前，SparkR 的 DataFrame 已经支持非常广泛的数据源，包括我们常用的 HDFS 文件系统、JSON 文件、CSV、Hive 数据等，同时也可以通过 JDBC 读取关系型数据库或者通过 Spark SQL 读取外部数据源。

我们可以通过两种方式使用 SparkR。

√ 在操作系统终端环境下输入“sparkR”命令启动一个 sparkR shell 环境，系统会自动为我们构建好 SparkContent 和 SQLContent，然后我们直接编写要运行的程序代码即可。

√ 如果是在程序中使用 sparkR，我们需要使用 sparkR. init 来构建 SparkContext，再将我们要用的其他配置选项传入给它，例如“sc <- sparkR. init()”。如果我们要使用 DataFrame，则通过 SparkContext 构造出 SQLContext 后即可使用，例如“sqlContext <- sparkRSQL. init(sc)”。

下面我们以使用 SparkR shell 为例，来展示下 SparkR 的用法。在操作系统的命令行界面输入“sparkR”命令后，我们依次执行以下代码。

SparkR shell 使用示例

```
1: > t <- read.df(sqlContext, "/logs/accessLog", "json")
2: > printSchema(t)
   root
    |-- userid:string(nullable = false)
    |-- timestamp:double(nullable = true)
    |-- rafficUp:double(nullable = true)
    |-- trafficDown:double(nullable = true)
3: > head(t)
     userid  timestamp  trafficUp  trafficDown
   1     71  1453113613      6229        27316
   2     53  1453113613      8441         8440
   3     57  1453113613      7462        82541
   4     63  1453113613      1119        41182
   5     83  1453113613      5381        21903
   6     37  1453113613      9231        81639
4: > registerTempTable(t, "table1")
5: > res <- sql(sqlContext, "SELECT count(*) FROM table1")
6: > head(res)
         _c0
   1  378859
7: > hiveContext <- sparkRHive.init(sc)
8: > t <- sql(hiveContext, "show tables")
```

```
9:  > showDF(t)

+--------------------+-----------+
|           tablename|isTemporary|
+--------------------+-----------+
|                 aaa|      false|
|           aisinoyjy|      false|
|                 bbb|      false|
|          cb_fpcg1_mx|      false|
|cb_fpcgl_mx_to_mysql1|      false|
|                 ccc|      false|
|        complain_log|      false|
|           customers|      false|
|          customtest|      false|
|                 ddd|      false|
|                 doc|      false|
|                docs|      false|
|                 eee|      false|
|     meta_keyattr_der|      false|
|               pokes|      false|
|       pokes_praquet|      false|
|           pokesnow|      false|
|             regionl|      false|
|           regionnum|      false|
|           sample_07|      false|
+--------------------+-----------+
    only showing top 20 rows
```

代码第 1 行读取一个以 json 格式存储的日志文件 accessLog。代码第 2 行以树形格式输出数据结构。代码第 3 行查看文件的数据结构定义和前面的部分数据。代码第 4 行将读取的数据注册成为一个临时表 table1 。代码第 5 行统计 table1 表中有多少条记录,并存储在变量 res 中。代码第 6 行输出统计结果。代码第 7 行生成一个 hiveContext 对象以使用 DataFrame。代码第 8 行调用显示数据表的命令,并在代码第 9 行输出所有的数据表。

参 考 文 献

[1] Moore G E. Progress in digital integrated electronics[C]. Electron Devices Meeting, 1975,21: 11-13.

[2] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: bringing order to the web[J]. Technical report,Stanford Digital Library Technologies Project,1998.

[3] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]. ACM SIGOPS operating systems review, 2003, 37(5):29-43.

[4] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1):107-113.

[5] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):4.

[6] NDFS, http:// wiki. apache. org/ nutch/ NutchDistributedFileSystem [OL]. 2016-06-01.

[7] Lloyd S P. Least squares quantization in PCM[J]. Information Theory, 1982, 28(2):129-137.

[8] 刘鹏, 滕家雨, 张国鹏, 等. 基于 Spark 的大规模文本 k-means 并行聚类算法[Z]. 北京: 中国大数据技术大会, 2014.

[9] Zaharia M A. An Architecture for and Fast and General Data Processing on Large Clusters [Z]. 2013.

[10] Thulasiraman K, Swamy MN S. Acyclic Directed Graphs[M]. Graphs: Theory and Algorithms. United States: Wiley-interscience, 1992:118.

[11] Zaharia M,Chowdhury M, Das T,et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012.

[12] Miner D, Shook A. MapReduce design patterns: Building effective algorithms and analytics for hadoop and other systems[M]. United States: O'Reilly Media, Inc. , 2012.

[13] Bengio Y, Schwenk H, Senécal JS, et al. Neural probabilistic language models[J]. Innovations in Machine Learning. Springer Berlin Heidelberg, 2006, 194(10):137-186.

[14] Mnih A, Teh Y W. A fast and simple algorithm for training neural probabilistic language models [C]. Proceeding of the international conference on Machine Learning, UK, 2012.

[15] Goh K I, Cusick M E, Valle D, et al. The human disease network[J]. Proceedings of the National Academy of Sciences,2007,104(21):8685-8690.

[16] Myers J L, Well A, Lorch R F. Research design and statistical analysis [M]. New York: Routledge, 2010.

[17] Kendall M G. A new measure of rank correlation[J]. Biometrika,1938, 30(1/ 2):81-93.

[18] Karau H, Konwinski A, Wendell P. Learning spark: Lightning-fast big data analysis[M]. United States: O'Reilly Media, Inc. , 2015.

[19] Babcock B ,Olston C. Distributed top-k monitoring[C]. Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003:28-39.

[20] MacQueen J. Some methods for classification and analysis of multivariate observations [C]. Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, 1967, 1 (14):281-297.

[21] Zhao W, Ma H, He Q. Parallel k-means clustering based on mapreduce[C]. Cloud computing.

Springer Berlin Heidelberg, 2009,5 931(1):674-679.

[22] Han J, Kamber M, Pei J. Data mining: Concepts and techniques [M]. United States: Elsevier, 2011.

[23] Altman N S. An introduction to kernel and nearest-neighbor nonparametric regression [J]. The American Statistician, 1992, 46(3):175-185.

[24] Russell S, Norvig P. Artificial Intelligence: A modern approach [M]. Upper Saddle River: Prentice hall, 2003.